**Course: Computer Science Module: 6006CEM Machine Learning and Related Applications**

**Student Name: Saad Iftikhar**

**Student ID: 9789180**

**Coventry GitHub Repository URL** or **Coventry OneDrive URL** (mandatory):

< <https://github.coventry.ac.uk/iftikhars/9789180-SI-s1>>

Table of Contents

[Table of figures 2](#_Toc88977949)

[Introduction 3](#_Toc88977950)

[**Problem statement**: Suicide risk prediction - Classification problem 3](#_Toc88977951)

[Motivation: 3](#_Toc88977952)

[Related work 3](#_Toc88977953)

[Analysis and pre-processing 4](#_Toc88977954)

[Data analysis 4](#_Toc88977955)

[Taking care of missing data and outliers 4](#_Toc88977956)

[Encoding and checking correlation 4](#_Toc88977957)

[Handling unbalanced data 5](#_Toc88977958)

[Applying PCA, feature section and splitting the data into training and testing sets 5](#_Toc88977959)

[Implementation 6](#_Toc88977960)

[Version control 6](#_Toc88977961)

[Model selection 6](#_Toc88977962)

[Model optimization 6](#_Toc88977963)

[Results and Evaluation 7](#_Toc88977964)

[KNN 7](#_Toc88977965)

[Random Forest 7](#_Toc88977966)

[Decision Tree 7](#_Toc88977967)

[Conclusion 9](#_Toc88977968)

[References 10](#_Toc88977969)

[Appendix A 11](#_Toc88977970)

[Appendix B 12](#_Toc88977971)

[Appendix C 130](#_Toc88977972)

[Outlier analysis 130](#_Toc88977973)

[Appendix D 132](#_Toc88977974)

[Suicide Label Classification 132](#_Toc88977975)

[Appendix E 133](#_Toc88977976)

[Model optimization 133](#_Toc88977977)

[KNN 133](#_Toc88977978)

[KNN with hyperparameters optimized using different neighbours and weight set as distance 133](#_Toc88977979)

[Random Forest 136](#_Toc88977980)

[Decision Tree 141](#_Toc88977981)

# Table of figures

[Figure 1 Correlation matrix 4](#_Toc88977904)

[Figure 2 Label samples 5](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977905)

[Figure 3 Model Selection 6](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977906)

[Figure 4 Model Selection 6](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977907)

[Figure 5 f1 score 7](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977908)

[Figure 6 Overall model results 8](#_Toc88977909)

[Figure 8 Population 130](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977910)

[Figure 9 Population 130](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977911)

[Figure 10 Countries 130](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977912)

[Figure 11 population by age 130](#_Toc88977913)

[Figure 12 Suicide by population 131](#_Toc88977914)

[Figure 13 Refugee\_population\_by\_country\_or\_territory\_of\_origin by suicide 131](#_Toc88977915)

[Figure 14 GDP per year by suicide 131](#_Toc88977916)

[Figure 15 Suicide rate assessment 132](file:///C:\Users\User\Documents\6006CEM-9789180-SI-2122s1%20-%20full.docx#_Toc88977917)

[Figure 16 KNN Cross Validation 133](#_Toc88977918)

[Figure 17 KNN Random Search 133](#_Toc88977919)

[Figure 18 KNN Grid Search 133](#_Toc88977920)

[Figure 19 KNN Standardscalar 134](#_Toc88977921)

[Figure 20 KNN MinMax1 134](#_Toc88977922)

[Figure 21 KNN MinMax3 135](#_Toc88977923)

[Figure 22 KNN Normalizer 135](#_Toc88977924)

[Figure 23 Random Forest Cross Validation 1 136](#_Toc88977925)

[Figure 24 Random Forest Cross Validation 2 136](#_Toc88977926)

[Figure 25 Random Forest Cross Validation 3 137](#_Toc88977927)

[Figure 26 Random Forest Cross Validation 4 137](#_Toc88977928)

[Figure 27 Random Forest Cross Validation 5 137](#_Toc88977929)

[Figure 28 Random Forest Cross Validation 6 138](#_Toc88977930)

[Figure 29 Random forest default 138](#_Toc88977931)

[Figure 30 Random forest StandardScalar 139](#_Toc88977932)

[Figure 31 Random forest MinMaxScalar 139](#_Toc88977933)

[Figure 32 Random forest RobustScalar 140](#_Toc88977934)

[Figure 33 Random forest Normalizer 140](#_Toc88977935)

[Figure 34 Decision Tree with cross validation 1 141](#_Toc88977936)

[Figure 35 Decision Tree with cross validation 2 141](#_Toc88977937)

[Figure 36 Decision Tree with cross validation 3 141](#_Toc88977938)

[Figure 37 Decision Tree with cross validation 4 142](#_Toc88977939)

[Figure 38 Decision Tree with cross validation 5 142](#_Toc88977940)

[Figure 39 Decision Tree with cross validation 6 142](#_Toc88977941)

[Figure 40 Decision Tree with cross validation 7 143](#_Toc88977942)

[Figure 41 Decision Tree with cross validation 8 143](#_Toc88977943)

[Figure 42 Decision Tree with Random Search 143](#_Toc88977944)

[Figure 43 Decision Tree with Grid Search 144](#_Toc88977945)

[Figure 44 Decision Tree Default 144](#_Toc88977946)

[Figure 45 Decision Tree StandardScalar 145](#_Toc88977947)

[Figure 46 Decision Tree MinMaxScalar 145](#_Toc88977948)

# Introduction

## **Problem statement**: Suicide risk prediction - Classification problem

## Motivation:

In research from [Brådvik (2018, p. 1)](#Brådvik) the suicide mortality rate is 1.4 percent of all global deaths. Biological, psychological, clinical, social, and environmental variables all have a role in the development of suicide risk. Because several risk variables are involved in defining a person's risk of suicide, determining a person's risk of suicide is difficult. Using computer testing and genetic screening, for example, to improve suicide risk assessment is a topic of continuing study [(Turecki et al., 2019, pp. 1)](#Turecki). To lower the number of suicide fatalities, prevention is essential [(Turecki et al., 2019, pp. 1)](#Turecki). This project aims to predict the risk factor of a person committing suicide so it could be prevented using a machine learning model.

## Related work

Previous work on risk suicide detection was looked at when choosing the dataset. The Study from [(NIROOMAND, 2020)](#Niroomand_work) carries a similarity to this study in that both studies focus on using classification machine learning algorithms to investigate probable factors that may raise the probability of suicide in people and try to predict the risk of suicide by using the same dataset. Both their study and this one uses the same machine learning algorithms except logistic regression to determine the required outcome.

This research determines the best algorithms to use from a selection of algorithms which are then further improved to predict the best results by hyperparameter optimization and feature scaling which was not done in the research performed by [(NIROOMAND, 2020)](#Niroomand_work). This study also uses multiclass classification rather than binary classification.

# Analysis and pre-processing

## Data analysis

The dataset used in this project was obtained from kaggle.com and it is called “suicidedataextrafestures” it was collected by the Kaggle user DORNA NIROOMAND [(Niroomand, 2019)](#Niroomand_data). The dataset has 26 columns containing 5 categorical and 21 continuous variables. The data is collected from the year 1985 to 2016 from 48 different countries. The data is from 2 sexes, male and female and 6 different age groups. There are 17 columns in the dataset with 15110 rows; the rest of the columns have rows of different lengths. Outliers and null values are present in the dataset.

The dataset used in this project is available online at: <https://www.kaggle.com/dornani/widandsuicide?select=suicidedataextrafestures.csv>

## Taking care of missing data and outliers

Firstly, the data frame is imported, columns with high nan values and columns with repeated information are removed. Nan values are again checked from the remaining column and are replaced using the mean of the 10 K nearest neighbours which were determined as the best course of action for this case after researching [(Maladkar, 2018)](#Maladkar). The index of the data frame is reset and then outliers are checked. An outlier can be kept, removed or winsorized using any method that could create a high bias [Yang and Berdine (2016, p. 52-56)](#Yang). Outliers can capture information that may be a vital part of the area of study (Frost, 2021). When removing outlines, it's best to look at how they affect the subject area [(Frost, 2021)](#Frost). After observation, it was determined that the outliers are providing valuable information [(see Appendix C)](#_Appendix_C) and hence they will not be removed or winsorized.

## Encoding and checking correlation

Now all columns of type objects containing categorical features are one hot encoded except the country column. A heat map is plotted (see Figure 1 Correlation matrix) so the highest correlated features can be identified and be assigned as a new data frame to a variable. They will be used later to test whether they improve the model’s performance or not.

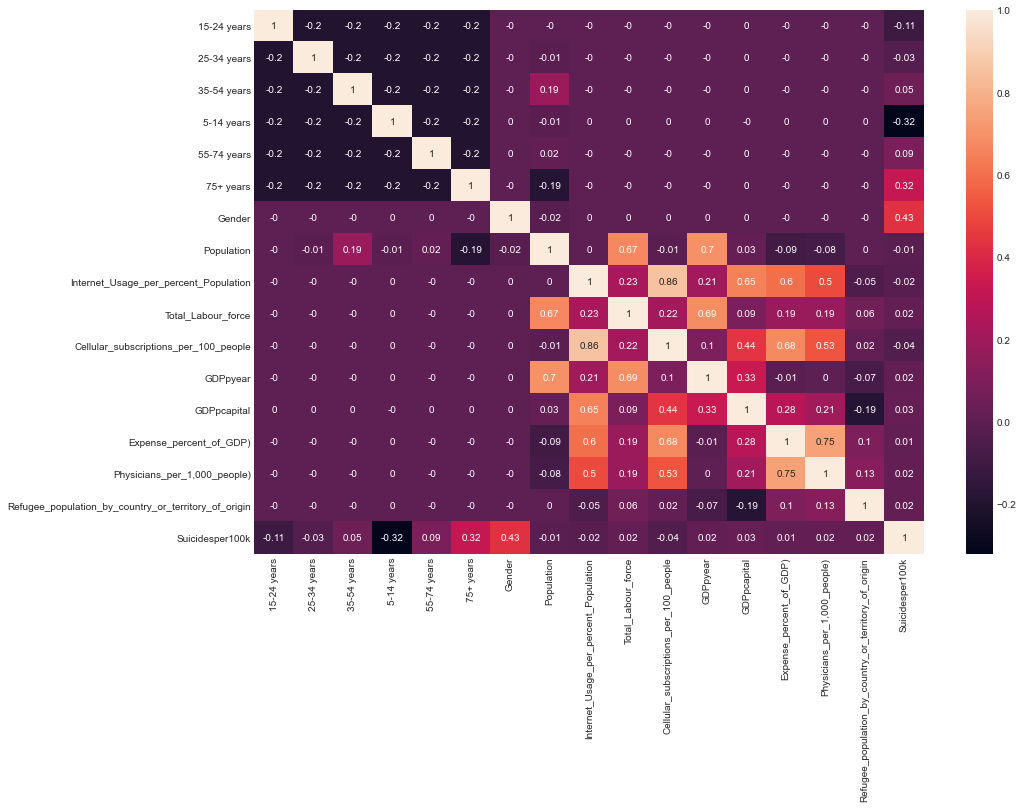


Figure Correlation matrix

The country column is one hot encoded and after the encoding, there are a total of 64 columns in the dataset. Labels 1, 2, 3, 4, 5 are made using the sucidesper100k column. 1 being minimum risk, 2 being low risk, 3 being medium risk, 4 being high risk and 5 being a maximum risk for more on this [(see Appendix D)](#_Appendix_D). This was done so there are enough categories to model and the model is not too computationally heavy.

## Handling unbalanced data

The label samples are observed for each of the newly created label classes that will be used to perform oversampling. As it can be seen in (Figure 2 Label samples) that label samples are imbalanced. 1 having 9264 samples comprises 61.31% of the data, 2 having 2850 samples covers 18.86 % of the data 3 having 2177 samples covers 14.41 % of the data, 4 having 678 samples covers 4.49 % of the data and 5 having 141 samples covers 0.93 % of the total data in the label column suicideper100k.

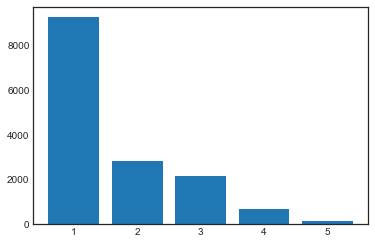


Figure Label samples

## Applying PCA, feature section and splitting the data into training and testing sets

The features x and X2 are selected using the original dataset and the dataset obtained after using correlation respectively. The label y is used for both features. Principal component analysis is applied to all 9 continuous features in the dataset. PCA should not be applied to categorical data that has been one hot encoded or to data, not on the coordinate plane [(Walker, 2019)](#Walker). After applying PCA it was determined that the components could be reduced to 6 dimensions for above 90 % of data to be kept, so it was decided that PCA would not have a great impact on the model evaluation, and they were not utilized for model evaluation. An Oversampled label is y\_over selected and a complementary X\_over feaure is also created so that results from the original features X, features selected using correlation matrix X2 and features X\_over, could be applied to different machine learning algorithms and the results could be evaluated.

# Implementation

## Version control

Version control was implemented using Github throughout his project and it can be found online at: <https://github.coventry.ac.uk/iftikhars/9789180-SI-s1>

## Model selection

Supervised classification learning algorithms will be performed in this project. Cross-validation was performed on Logistic, KNN (K nearest neighbours), Naive Bayes, Neural Networks, Decision tree and Random Forest models. It was decided that the Decision Tree with a score of 93 %, Random forest scoring 94 % and KNN having a score of 71 % had the best cross-validated accuracy. The tests were performed on all three X, X2 and X\_over features with there complementary y variables and the best results were obtained from the oversampled features. These results can be observed in Figure 3 Model Selection and Figure 4 Model Selection. SVM and XGBoost were not used in this project because of hardware requirements and time constrains. The features giving the best performance are split into testing and training sets. It is split 80 % into the training and 20 % into the testing sets with the random state set to 1.

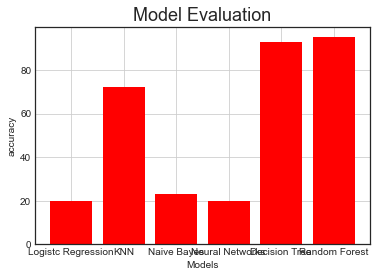


Figure Model Selection

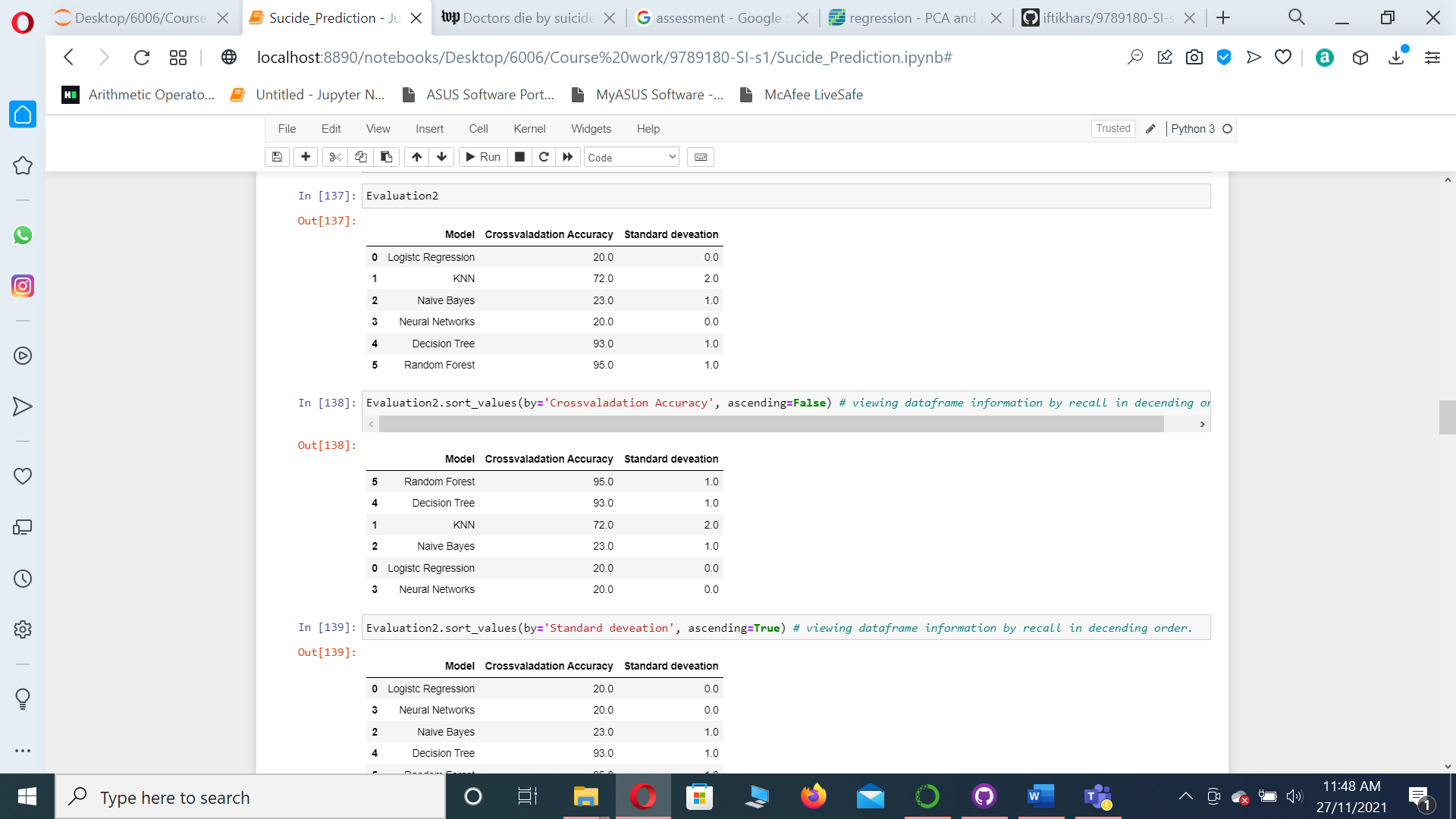


Figure Model Selection

## Model optimization

Different hyperparameter optimization techniques were applied to the three selected models. They were applied using the following logic and order. First cross-validation was applied to narrow down on which hyperparameters give the best results individually, then random search and grid search was applied to a range of hyperparameters that had the most effect on increasing the accuracy of the model. Accuracy was chosen as a measure to optimize the hyperparameters as it was important how many correct predictions the model made for the test dataset.

After the hyperparameter selection. All the models were applied on selected features and the continuous variables of the selected features were feature scaled using Standardscaler, MinMaxscaler, Robustscaler and Normaliser feature scaling techniques.

# Results and Evaluation

## KNN

KNN had the best testing and training set results with hyperparameters, 3 neighbours, weight set as distance using MinMaxScalar and it had similar overall scores using other scaling techniques, but the AUC (Area under the curve) was highest using the MinMaxScalar with 3 neighbours (see Figure 21 KNN MinMax3). When evaluated with cross-validation it performed best using 15 neighbours, weight set as distance with 80.22 % accuracy and 1.40 % standard deviation (see Appendix E).

## Random Forest

Random Forest had the best overall results on the testing and training sets with 1 min\_samples\_leaf, 2 min\_samples\_split, None max\_leaf\_nodes, 0.0 min\_weight\_fraction\_leaf, 'entropy' criterion, None max\_features, 121 max\_depth, 'random' splitter as hyperparameters score with MinMaxScalar and it had similar overall scores using other scaling techniques, but the AUC (Area under the curve) was highest using the MinMaxScalar (see Figure 46 Decision Tree MinMaxScalar). The defaults model results were 93.11 % with a 0.86 % standard deviation when evaluated with cross-validation (see Appendix E).

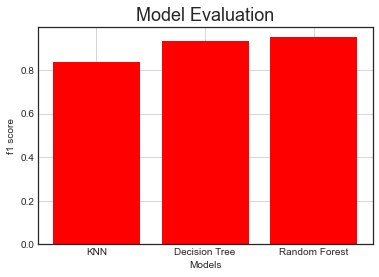
## Decision Tree

Decision Tree has the best overall results with 1 min\_samples\_leaf, 2 min\_samples\_split, None max\_leaf\_nodes, 0.0 min\_weight\_fraction\_leaf, 'entropy' criterion, None max\_features, 121 max\_depth, 'random' splitter as hyperparameters with MinMaxScalar (see Figure 46 Decision Tree MinMaxScalar ) has the best area under the curve. The defaults model results were 93.11 % with a 0.86 % standard deviation. The results after using MinMaxScaler with hyperparameter optimization were 94.44 % with a 1.05 standard deviation (see Appendix E).

.

Because the algorithms are estimating people's probability of suicide, how many times they are predicted accurately is more significant than predicting anything positive, and how many times they were actually positive recall will be a top priority. It will not be the exclusive method of scoring since a model with 100% recall may always make a positive prediction; it would have 100% recall but be entirely useless as a result, the f1 score will be used to evaluate the model because it prioritises recall and accuracy equally. As seen in and Figure 5 f1 score the Random Forest model has the highest performance when predicting the f1 score with a mean f1 score of 0.950965 and the mean accuracy is 0.956131 with 10 cross-validations.

Figure f1 score



Random Forest with hyperparameters 1 min\_samples\_leaf ,2 min\_samples\_split , None max\_leaf\_nodes , 0.0 min\_weight\_fraction\_leaf , 'entropy' criterion, None max\_features, 121 max\_depth, 'random' splitter gave 0.956131 accuracy, 0.961529 precision, 0.954255 recall, 0.950965 f1 score and 0.998497 Area under the curve when run with 10 cross-validations using on oversampled labels and features scaled using MinMaxScalar. When subtracting the standard deviation from (Figure 6 Overall model results) the Random Forest model gives 0.9448089 accuracy, 0.953715 precision, 0.931089 recall, 0.937287 f1 score and 0.997383 Area under the curve which overall higher scores than KNN and Decision Tree Models.

Graphical user interface, text, application

Description automatically generated

Figure Overall model results

# Conclusion

In Conclusion, the Random Forest model with hyperparameter with hyperparameters 1 min\_samples\_leaf, 2 min\_samples\_split, None max\_leaf\_nodes, 0.0 min\_weight\_fraction\_leaf, 'entropy' criterion, None max\_features, 121 max\_depth, 'random' splitter applied on oversampled labels and features scaled by using MinMaxScalar gives the best prediction for suicide.

# References

Brådvik, L. (2018). Suicide Risk and Mental Disorders. *International Journal of Environmental Research and Public Health*, *15*(9), 1–4. <https://doi.org/10.3390/ijerph15092028>

Belsher, B. E., Smolenski, D. J., Pruitt, L. D., Bush, N. E., Beech, E. H., Workman, D. E., Morgan, R. L., Evatt, D. P., Tucker, J., & Skopp, N. A. (2019). Prediction Models for Suicide Attempts and Deaths. *JAMA Psychiatry*, *76*(6), 642–651. <https://doi.org/10.1001/jamapsychiatry.2019.0174>

Frost, J. (2021, April 5). Guidelines for Removing and Handling Outliers in Data. Statistics By Jim. Retrieved 28 November 2021, from <https://statisticsbyjim.com/basics/remove-outliers/>

Maladkar, K. (2021, October 7). 5 Ways To Handle Missing Values In Machine Learning Datasets. Analytics India Magazine. Retrieved 28 November 2021, from <https://analyticsindiamag.com/5-ways-handle-missing-values-machine-learning-datasets/>

Niroomand, D. (2020, January 31). A classification analysis on suicide data. Kaggle. Retrieved 28 November 2021, from <https://www.kaggle.com/dornani/a-classification-analysis-on-suicide-data>

Niroomand, D. (2019, November 11). Suicide rate and WDI data. Kaggle. Retrieved 28 November 2021, from <https://www.kaggle.com/dornani/widandsuicide?select=suicidedataextrafestures.csv>

Turecki, G., Brent, D. A., Gunnell, D., O’Connor, R. C., Oquendo, M. A., Pirkis, J., & Stanley, B. H. (2019). Suicide and suicide risk. *Nature reviews Disease primers*, *5*(1), 1-22.

Walker, B. (2019, December 31). PCA Is Not Feature Selection - Towards Data Science. Medium. Retrieved 28 November 2021, from <https://towardsdatascience.com/pca-is-not-feature-selection-3344fb764ae6>

Yang, S., & Berdine, G. (2016). Outliers. *The Southwest Respiratory and Critical Care Chronicles*, *4*(13), 52-56.

# Appendix A

<

**Coventry** GitHub Repository URL: <https://github.coventry.ac.uk/iftikhars/9789180-SI-s1>

The Demonstration video is present in the Coventry university github and is available online at: <https://github.coventry.ac.uk/iftikhars/9789180-SI-s1>

The dataset used in this project is available on at: <https://www.kaggle.com/dornani/widandsuicide?select=suicidedataextrafestures.csv>

The source code is present in the [Appendix B](#_Appendix_B).

>

# Appendix B

< 1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

474

475

476

477

478

479

480

481

482

483

484

485

486

487

488

489

490

491

492

493

494

495

496

497

498

499

500

501

502

503

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561

562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

595

596

597

598

599

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

619

620

621

622

623

624

625

626

627

628

629

630

631

632

633

634

635

636

637

638

639

640

641

642

643

644

645

646

647

648

649

650

651

652

653

654

655

656

657

658

659

660

661

662

663

664

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684

685

686

687

688

689

690

691

692

693

694

695

696

697

698

699

700

701

702

703

704

705

706

707

708

709

710

711

712

713

714

715

716

717

718

719

720

721

722

723

724

725

726

727

728

729

730

731

732

733

734

735

736

737

738

739

740

741

742

743

744

745

746

747

748

749

750

751

752

753

754

755

756

757

758

759

760

761

762

763

764

765

766

767

768

769

770

771

772

773

774

775

776

777

778

779

780

781

782

783

784

785

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

834

835

836

837

838

839

840

841

842

843

844

845

846

847

848

849

850

851

852

853

854

855

856

857

858

859

860

861

862

863

864

865

866

867

868

869

870

871

872

873

874

875

876

877

878

879

880

881

882

883

884

885

886

887

888

889

890

891

892

893

894

895

896

897

898

899

900

901

902

903

904

905

906

907

908

909

910

911

912

913

914

915

916

917

918

919

920

921

922

923

924

925

926

927

928

929

930

931

932

933

934

935

936

937

938

939

940

941

942

943

944

945

946

947

948

949

950

951

952

953

954

955

956

957

958

959

960

961

962

963

964

965

966

967

968

969

970

971

972

973

974

975

976

977

978

979

980

981

982

983

984

985

986

987

988

989

990

991

992

993

994

995

996

997

998

999

1000

1001

1002

1003

1004

1005

1006

1007

1008

1009

1010

1011

1012

1013

1014

1015

1016

1017

1018

1019

1020

1021

1022

1023

1024

1025

1026

1027

1028

1029

1030

1031

1032

1033

1034

1035

1036

1037

1038

1039

1040

1041

1042

1043

1044

1045

1046

1047

1048

1049

1050

1051

1052

1053

1054

1055

1056

1057

1058

1059

1060

1061

1062

1063

1064

1065

1066

1067

1068

1069

1070

1071

1072

1073

1074

1075

1076

1077

1078

1079

1080

1081

1082

1083

1084

1085

1086

1087

1088

1089

1090

1091

1092

1093

1094

1095

1096

1097

1098

1099

1100

1101

1102

1103

1104

1105

1106

1107

1108

1109

1110

1111

1112

1113

1114

1115

1116

1117

1118

1119

1120

1121

1122

1123

1124

1125

1126

1127

1128

1129

1130

1131

1132

1133

1134

1135

1136

1137

1138

1139

1140

1141

1142

1143

1144

1145

1146

1147

1148

1149

1150

1151

1152

1153

1154

1155

1156

1157

1158

1159

1160

1161

1162

1163

1164

1165

1166

1167

1168

1169

1170

1171

1172

1173

1174

1175

1176

1177

1178

1179

1180

1181

1182

1183

1184

1185

1186

1187

1188

1189

1190

1191

1192

1193

1194

1195

1196

1197

1198

1199

1200

1201

1202

1203

1204

1205

1206

1207

1208

1209

1210

1211

1212

1213

1214

1215

1216

1217

1218

1219

1220

1221

1222

1223

1224

1225

1226

1227

1228

1229

1230

1231

1232

1233

1234

1235

1236

1237

1238

1239

1240

1241

1242

1243

1244

1245

1246

1247

1248

1249

1250

1251

1252

1253

1254

1255

1256

1257

1258

1259

1260

1261

1262

1263

1264

1265

1266

1267

1268

1269

1270

1271

1272

1273

1274

1275

1276

1277

1278

1279

1280

1281

1282

1283

1284

1285

1286

1287

1288

1289

1290

1291

1292

1293

1294

1295

1296

1297

1298

1299

1300

1301

1302

1303

1304

1305

1306

1307

1308

1309

1310

1311

1312

1313

1314

1315

1316

1317

1318

1319

1320

1321

1322

1323

1324

1325

1326

1327

1328

1329

1330

1331

1332

1333

1334

1335

1336

1337

1338

1339

1340

1341

1342

1343

1344

1345

1346

1347

1348

1349

1350

1351

1352

1353

1354

1355

1356

1357

1358

1359

1360

1361

1362

1363

1364

1365

1366

1367

1368

1369

1370

1371

1372

1373

1374

1375

1376

1377

1378

1379

1380

1381

1382

1383

1384

1385

1386

1387

1388

1389

1390

1391

1392

1393

1394

1395

1396

1397

1398

1399

1400

1401

1402

1403

1404

1405

1406

1407

1408

1409

1410

1411

1412

1413

1414

1415

1416

1417

1418

1419

1420

1421

1422

1423

1424

1425

1426

1427

1428

1429

1430

1431

1432

1433

1434

1435

1436

1437

1438

1439

1440

1441

1442

1443

1444

1445

1446

1447

1448

1449

1450

1451

1452

1453

1454

1455

1456

1457

1458

1459

1460

1461

1462

1463

1464

1465

1466

1467

1468

1469

1470

1471

1472

1473

1474

1475

1476

1477

1478

1479

1480

1481

1482

1483

1484

1485

1486

1487

1488

1489

1490

1491

1492

1493

1494

1495

1496

1497

1498

1499

1500

1501

1502

1503

1504

1505

1506

1507

1508

1509

1510

1511

1512

1513

1514

1515

1516

1517

1518

1519

1520

1521

1522

1523

1524

1525

1526

1527

1528

1529

1530

1531

1532

1533

1534

1535

1536

1537

1538

1539

1540

1541

1542

1543

1544

1545

1546

1547

1548

1549

1550

1551

1552

1553

1554

1555

1556

1557

1558

1559

1560

1561

1562

1563

1564

1565

1566

1567

1568

1569

1570

1571

1572

1573

1574

1575

1576

1577

1578

1579

1580

1581

1582

1583

1584

1585

1586

1587

1588

1589

1590

1591

1592

1593

1594

1595

1596

1597

1598

1599

1600

1601

1602

1603

1604

1605

1606

1607

1608

1609

1610

1611

1612

1613

1614

1615

1616

1617

1618

1619

1620

1621

1622

1623

1624

1625

1626

1627

1628

1629

1630

1631

1632

1633

1634

1635

1636

1637

1638

1639

1640

1641

1642

1643

1644

1645

1646

1647

1648

1649

1650

1651

1652

1653

1654

1655

1656

1657

1658

1659

1660

1661

1662

1663

1664

1665

1666

1667

1668

1669

1670

1671

1672

1673

1674

1675

1676

1677

1678

1679

1680

1681

1682

1683

1684

1685

1686

1687

1688

1689

1690

1691

1692

1693

1694

1695

1696

1697

1698

1699

1700

1701

1702

1703

1704

1705

1706

1707

1708

1709

1710

1711

1712

1713

1714

1715

1716

1717

1718

1719

1720

1721

1722

1723

1724

1725

1726

1727

1728

1729

1730

1731

1732

1733

1734

1735

1736

1737

1738

1739

1740

1741

1742

1743

1744

1745

1746

1747

1748

1749

1750

1751

1752

1753

1754

1755

1756

1757

1758

1759

1760

1761

1762

1763

1764

1765

1766

1767

1768

1769

1770

1771

1772

1773

1774

1775

1776

1777

1778

1779

1780

1781

1782

1783

1784

1785

1786

1787

1788

1789

1790

1791

1792

1793

1794

1795

1796

1797

1798

1799

1800

1801

1802

1803

1804

1805

1806

1807

1808

1809

1810

1811

1812

1813

1814

1815

1816

1817

1818

1819

1820

1821

1822

1823

1824

1825

1826

1827

1828

1829

1830

1831

1832

1833

1834

1835

1836

1837

1838

1839

1840

1841

1842

1843

1844

1845

1846

1847

1848

1849

1850

1851

1852

1853

1854

1855

1856

1857

1858

1859

1860

1861

1862

1863

1864

1865

1866

1867

1868

1869

1870

1871

1872

1873

1874

1875

1876

1877

1878

1879

1880

1881

1882

1883

1884

1885

1886

1887

1888

1889

1890

1891

1892

1893

1894

1895

1896

1897

1898

1899

1900

1901

1902

1903

1904

1905

1906

1907

1908

1909

1910

1911

1912

1913

1914

1915

1916

1917

1918

1919

1920

1921

1922

1923

1924

1925

1926

1927

1928

1929

1930

1931

1932

1933

1934

1935

1936

1937

1938

1939

1940

1941

1942

1943

1944

1945

1946

1947

1948

1949

1950

1951

1952

1953

1954

1955

1956

1957

1958

1959

1960

1961

1962

1963

1964

1965

1966

1967

1968

1969

1970

1971

1972

1973

1974

1975

1976

1977

1978

1979

1980

1981

1982

1983

1984

1985

1986

1987

1988

1989

1990

1991

1992

1993

1994

1995

1996

1997

1998

1999

2000

2001

2002

2003

2004

2005

2006

2007

2008

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019

2020

2021

2022

2023

2024

2025

2026

2027

2028

2029

2030

2031

2032

2033

2034

2035

2036

2037

2038

2039

2040

2041

2042

2043

2044

2045

2046

2047

2048

2049

2050

2051

2052

2053

2054

2055

2056

2057

2058

2059

2060

2061

2062

2063

2064

2065

2066

2067

2068

2069

2070

2071

2072

2073

2074

2075

2076

2077

2078

2079

2080

2081

2082

2083

2084

2085

2086

2087

2088

2089

2090

2091

2092

2093

2094

2095

2096

2097

2098

2099

2100

2101

2102

2103

2104

2105

2106

2107

2108

2109

2110

2111

2112

2113

2114

2115

2116

2117

2118

2119

2120

2121

2122

2123

2124

2125

2126

2127

2128

2129

2130

2131

2132

2133

2134

2135

2136

2137

2138

2139

2140

2141

2142

2143

2144

2145

2146

2147

2148

2149

2150

2151

2152

2153

2154

2155

2156

2157

2158

2159

2160

2161

2162

2163

2164

2165

2166

2167

2168

2169

2170

2171

2172

2173

2174

2175

2176

2177

2178

2179

2180

2181

2182

2183

2184

2185

2186

2187

2188

2189

2190

2191

2192

2193

2194

2195

2196

2197

2198

2199

2200

2201

2202

2203

2204

2205

2206

2207

2208

2209

2210

2211

2212

2213

2214

2215

2216

2217

2218

2219

2220

2221

2222

2223

2224

2225

2226

2227

2228

2229

2230

2231

2232

2233

2234

2235

2236

2237

2238

2239

2240

2241

2242

2243

2244

2245

2246

2247

2248

2249

2250

2251

2252

2253

2254

2255

2256

2257

2258

2259

2260

2261

2262

2263

2264

2265

2266

2267

2268

2269

2270

2271

2272

2273

2274

2275

2276

2277

2278

2279

2280

2281

2282

2283

2284

2285

2286

2287

2288

2289

2290

2291

2292

2293

2294

2295

2296

2297

2298

2299

2300

2301

2302

2303

2304

2305

2306

2307

2308

2309

2310

2311

2312

2313

2314

2315

2316

2317

2318

2319

2320

2321

2322

2323

2324

2325

2326

2327

2328

2329

2330

2331

2332

2333

2334

2335

2336

2337

2338

2339

2340

2341

2342

2343

2344

2345

2346

2347

2348

2349

2350

2351

2352

2353

2354

2355

2356

2357

2358

2359

2360

2361

2362

2363

2364

2365

2366

2367

2368

2369

2370

2371

2372

2373

2374

2375

2376

2377

2378

2379

2380

2381

2382

2383

2384

2385

2386

2387

2388

2389

2390

2391

2392

2393

2394

2395

2396

2397

2398

2399

2400

2401

2402

2403

2404

2405

2406

2407

2408

2409

2410

2411

2412

2413

2414

2415

2416

2417

2418

2419

2420

2421

2422

2423

2424

2425

2426

2427

2428

2429

2430

2431

2432

2433

2434

2435

2436

2437

2438

2439

2440

2441

2442

2443

2444

2445

2446

2447

2448

2449

2450

2451

2452

2453

2454

2455

2456

2457

2458

2459

2460

2461

2462

2463

2464

2465

2466

2467

2468

2469

2470

2471

2472

2473

2474

2475

2476

2477

2478

2479

2480

2481

2482

2483

2484

2485

2486

2487

2488

2489

2490

2491

2492

2493

2494

2495

2496

2497

2498

2499

2500

2501

2502

2503

2504

2505

2506

2507

2508

2509

2510

2511

2512

2513

2514

2515

2516

2517

2518

2519

2520

2521

2522

2523

2524

2525

2526

2527

2528

2529

2530

2531

2532

2533

2534

2535

2536

2537

2538

2539

2540

2541

2542

2543

2544

2545

2546

2547

2548

2549

2550

2551

2552

2553

2554

2555

2556

2557

2558

2559

2560

2561

2562

2563

2564

2565

2566

2567

2568

2569

2570

2571

2572

2573

2574

2575

2576

2577

2578

2579

2580

2581

2582

2583

2584

2585

2586

2587

2588

2589

2590

2591

2592

2593

2594

2595

2596

2597

2598

2599

2600

2601

2602

2603

2604

2605

2606

2607

2608

2609

2610

2611

2612

2613

2614

2615

2616

2617

2618

2619

2620

2621

2622

2623

2624

2625

2626

2627

2628

2629

2630

2631

2632

2633

2634

2635

2636

2637

2638

2639

2640

2641

2642

2643

2644

2645

2646

2647

2648

2649

2650

2651

2652

2653

2654

2655

2656

2657

2658

2659

2660

2661

2662

2663

2664

2665

2666

2667

2668

2669

2670

2671

2672

2673

2674

2675

2676

2677

2678

2679

2680

2681

2682

2683

2684

2685

2686

2687

2688

2689

2690

2691

2692

2693

2694

2695

2696

2697

2698

2699

2700

2701

2702

2703

2704

2705

2706

2707

2708

2709

2710

2711

2712

2713

2714

2715

2716

2717

2718

2719

2720

2721

2722

2723

2724

2725

2726

2727

2728

2729

2730

2731

2732

2733

2734

2735

2736

2737

2738

2739

2740

2741

2742

2743

2744

2745

2746

2747

2748

2749

2750

2751

2752

2753

2754

2755

2756

2757

2758

2759

2760

2761

2762

2763

2764

2765

2766

2767

2768

2769

2770

2771

2772

2773

2774

2775

2776

2777

2778

2779

2780

2781

2782

2783

2784

2785

2786

2787

2788

2789

2790

2791

2792

2793

2794

2795

2796

2797

2798

2799

2800

2801

2802

2803

2804

2805

2806

2807

2808

2809

2810

2811

2812

2813

2814

2815

2816

2817

2818

2819

2820

2821

2822

2823

2824

2825

2826

2827

2828

2829

2830

2831

2832

2833

2834

2835

2836

2837

2838

2839

2840

2841

2842

2843

2844

2845

2846

2847

2848

2849

2850

2851

2852

2853

2854

2855

2856

2857

2858

2859

2860

2861

2862

2863

2864

2865

2866

2867

2868

2869

2870

2871

2872

2873

2874

2875

2876

2877

2878

2879

2880

2881

2882

2883

2884

2885

2886

2887

2888

2889

2890

2891

2892

2893

2894

2895

2896

2897

2898

2899

2900

2901

2902

2903

2904

2905

2906

2907

2908

2909

2910

2911

2912

2913

2914

2915

2916

2917

2918

2919

2920

2921

2922

2923

2924

2925

2926

2927

2928

2929

2930

2931

2932

2933

2934

2935

2936

2937

2938

2939

2940

2941

2942

2943

2944

2945

2946

2947

2948

2949

2950

2951

2952

2953

2954

2955

2956

2957

2958

2959

2960

2961

2962

2963

2964

2965

2966

2967

2968

2969

2970

2971

2972

2973

2974

2975

2976

2977

2978

2979

2980

2981

2982

2983

2984

2985

2986

2987

2988

2989

2990

2991

2992

2993

2994

2995

2996

2997

2998

2999

3000

3001

3002

3003

3004

3005

3006

3007

3008

3009

3010

3011

3012

3013

3014

3015

3016

3017

3018

3019

3020

3021

3022

3023

3024

3025

3026

3027

3028

3029

3030

3031

3032

3033

3034

3035

3036

3037

3038

3039

3040

3041

3042

3043

3044

3045

3046

3047

3048

3049

3050

3051

3052

3053

3054

3055

3056

3057

3058

3059

3060

3061

3062

3063

3064

3065

3066

3067

3068

3069

3070

3071

3072

3073

3074

3075

3076

3077

3078

3079

3080

3081

3082

3083

3084

3085

3086

3087

3088

3089

3090

3091

3092

3093

3094

3095

3096

3097

3098

3099

3100

3101

3102

3103

3104

3105

3106

3107

3108

3109

3110

3111

3112

3113

3114

3115

3116

3117

3118

3119

3120

3121

3122

3123

3124

3125

3126

3127

3128

3129

3130

3131

3132

3133

3134

3135

3136

3137

3138

3139

3140

3141

3142

3143

3144

3145

3146

3147

3148

3149

3150

3151

3152

3153

3154

3155

3156

3157

3158

3159

3160

3161

3162

3163

3164

3165

3166

3167

3168

3169

3170

3171

3172

3173

3174

3175

3176

3177

3178

3179

3180

3181

3182

3183

3184

3185

3186

3187

3188

3189

3190

3191

3192

3193

3194

3195

3196

3197

3198

3199

3200

3201

3202

3203

3204

3205

3206

3207

3208

3209

3210

3211

3212

3213

3214

3215

3216

3217

3218

3219

3220

3221

3222

3223

3224

3225

3226

3227

3228

3229

3230

3231

3232

3233

3234

3235

3236

3237

3238

3239

3240

3241

3242

3243

3244

3245

3246

3247

3248

3249

3250

3251

3252

3253

3254

3255

3256

3257

3258

3259

3260

3261

3262

3263

3264

3265

3266

3267

3268

3269

3270

3271

3272

3273

3274

3275

3276

3277

3278

3279

3280

3281

3282

3283

3284

3285

3286

3287

3288

3289

3290

3291

3292

3293

3294

3295

3296

3297

3298

3299

3300

3301

## Downloading required libraries

# pip install imbalanced-learn # used to install imbalanced learn

# pip install xgboost # to install xgboost\_model

# Make sure the version of anaconda is the latest

**import** **sklearn** # importing sklearn

**print**(sklearn.\_\_version\_\_) # printing sklearn version

### Importing the relevent libraries

**import** **numpy** **as** **np** # Allows us to work with arrays.

**import** **pandas** **as** **pd** # importing pandas’ library for use. Allows us to import data set and manipulate it.

**import** **seaborn** **as** **sns** # Allows to polt beautiful plots.

**import** **matplotlib.pyplot** **as** **plt** # Allows working with plots.

**from** **mpl\_toolkits** **import** mplot3d # plotting 3d plots

**from** **sklearn.compose** **import** ColumnTransformer # helps with encoding.

**from** **sklearn.preprocessing** **import** OneHotEncoder # Does onehotencode.

**from** **sklearn.preprocessing** **import** LabelEncoder # Does 1 and 0 encoding.

**from** **sklearn.model\_selection** **import** train\_test\_split # Splits dataset into test set and traning set.

**from** **sklearn.preprocessing** **import** StandardScaler, MinMaxScaler, RobustScaler, Normalizer # Perform the feature scaling.

**from** **sklearn.metrics** **import** confusion\_matrix, ConfusionMatrixDisplay, multilabel\_confusion\_matrix # creates a

# confusion matrix # creates a confusion matrix

**from** **sklearn.metrics** **import** accuracy\_score # Returns accury score of a model.

**from** **collections** **import** Counter # Allows the counting the items in an iterable list.

**from** **sklearn.linear\_model** **import** LogisticRegression # Performs logistic regression.

**from** **sklearn.neighbors** **import** KNeighborsClassifier # applies K Neariesst Neighobour.

**from** **sklearn.neural\_network** **import** MLPClassifier # applies Multilayer Perceptrons neural network

**from** **sklearn.impute** **import** KNNImputer # imputes missing values using KNN.

**from** **sklearn.svm** **import** SVC # applies svm

**from** **sklearn.naive\_bayes** **import** GaussianNB # applies naive\_bayes gaussianNB.

**from** **sklearn.tree** **import** DecisionTreeClassifier # applies decision tree classification model.

**from** **sklearn.ensemble** **import** RandomForestClassifier # applies random forest classification.

**from** **xgboost** **import** XGBClassifier # applies xgboost classification. gradiant decision trees.

**from** **sklearn.decomposition** **import** PCA # applies PCA

**from** **sklearn.model\_selection** **import** cross\_val\_score # performs cross validation. Helps in model selection.

**from** **sklearn.model\_selection** **import** GridSearchCV # helps select the best hyper parameters

**from** **imblearn.over\_sampling** **import** RandomOverSampler # Uses over sampling techniques to Sample the data correctly.

**from** **sklearn.metrics** **import** classification\_report, accuracy\_score, recall\_score, precision\_score, roc\_auc\_score, f1\_score

# Allows the usage of a classification report

**from** **sklearn.metrics** **import** precision\_recall\_fscore\_support # gives precison, recall, f1 score, and support

**from** **sklearn.model\_selection** **import** RandomizedSearchCV # performs randomized search cv

**import** **warnings** # allows to ignore warnings

warnings.filterwarnings("ignore") # ignores warnings

#%matplotlib inline # helps in showing plots on the browser.

### Importing the dataframe

sucidedataframe = pd.read\_csv("suicidedataextrafestures.csv") # opens csv files and assighns them to a variable.

### Checking the data from the dataframe before pre-processing

sucidedataframe.head(**1**) # Taking a look at the dataframe the first elements of the dataset.

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

## 1. Data pre-processing

# Below relevent data is selected that will be used in this project.

sucidedataframe = sucidedataframe[["country", "age" ,"sex", "population", \

"Individuals using the Internet (% of population)", "Labor force, total", \

"Mobile cellular subscriptions (per 100 people)", "GDPpyear","GDPpcapital","Expense (% of GDP)",\

"Physicians (per 1,000 people)","Refugee population by country or territory of origin" ,"suicidesper100k"]]## 1. Data

# pre-processing

sucidedataframe.head(**5**) # out puts data from dataframe

# Below the col names are renamed.

sucidedataframe = sucidedataframe.set\_axis(["Country", "Age", "Gender", "Population", \

"Internet\_Usage\_per\_percent\_Population", "Total\_Labour\_force", "Cellular\_subscriptions\_per\_100\_people", \

"GDPpyear","GDPpcapital","Expense\_percent\_of\_GDP)",\

"Physicians\_per\_1,000\_people)","Refugee\_population\_by\_country\_or\_territory\_of\_origin" ,"Suicidesper100k"],axis=**1**)

sucidedataframe.columns # The columns of the dataframe are viewed.

sucidedataframe.shape # The Entries and the columns of the dataframe are viewed.

### Dealing with null values

sns.heatmap(sucidedataframe.isnull()) # shows null values

sucidedataframe.isnull().sum() # Checking the dataframe for null values.

sucidedataframe.info() # checking basic information on dataframe

#df['Internet\_Usage\_per\_percent\_Population'] = df['DataFrame Column'].fillna(mean(column))

**print**(len(sucidedataframe.Internet\_Usage\_per\_percent\_Population)) # printing lenghth of the column.

**print**(len(sucidedataframe.Refugee\_population\_by\_country\_or\_territory\_of\_origin)) # printing lenghth of the column.

**print**(len(sucidedataframe["Expense\_percent\_of\_GDP)"])) # printing lenghth of the column.

**print**(len(sucidedataframe["Physicians\_per\_1,000\_people)"])) # printing lenghth of the column.

# Function to replace missing values Using KNN

**def** **missng\_values\_filler\_knn**(missingdataframevalue,colname): # function replaces missing values using KNN's. function

# name specified and paramaters given

columntobereplaced = missingdataframevalue.to\_numpy() # The column to be replaced is stored in a variable.

imputer = KNNImputer(n\_neighbors=**10**, weights="uniform") # creating instance of the object.

replaced = imputer.fit\_transform(columntobereplaced) # fitting the model and replaing missing values

**print**(len(replaced)) # printing replaced lenght

dataframe=pd.DataFrame(replaced, columns=colname) # adding replaced values in a dataframe column and giving the column a

# name.

**print**("null values ", dataframe.isnull().sum()) # Checking the dataframe for null values.

**print**(dataframe.head(**5**)) # printing head of new data frame

**return** dataframe # returning the dataframe

# Replacing missing values

sucidedataframe[["Internet\_Usage\_per\_percent\_Population"]] = \

missng\_values\_filler\_knn(sucidedataframe[["Internet\_Usage\_per\_percent\_Population"]], ["Internet\_Usage\_per\_percent\_Population"])

# replacing null values using KNN function

sucidedataframe[["Expense\_percent\_of\_GDP)"]] = \

missng\_values\_filler\_knn(sucidedataframe[["Expense\_percent\_of\_GDP)"]], ["Expense\_percent\_of\_GDP)"])

# replacing null values using KNN function

sucidedataframe[["Refugee\_population\_by\_country\_or\_territory\_of\_origin"]] = \

missng\_values\_filler\_knn(sucidedataframe[["Refugee\_population\_by\_country\_or\_territory\_of\_origin"]],\

["Refugee\_population\_by\_country\_or\_territory\_of\_origin"])

# replacing null values using KNN function

sucidedataframe[["Physicians\_per\_1,000\_people)"]] = \

missng\_values\_filler\_knn(sucidedataframe[["Physicians\_per\_1,000\_people)"]],["Physicians\_per\_1,000\_people)"])

sucidedataframe.isnull().sum() # Checking the dataframe for null values.

sucidedataframe = sucidedataframe.dropna() # droping all rows with at least one null values.

# sucidedataframe.isnull().sum() # Checking the dataframe for null values.

sucidedataframe.shape # The Entries and the columns of the dataframe are viewed.

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

# Reseting index

sucidedataframe.reset\_index(drop=True, inplace=True) # reseting index.

### Looking at outliers.

**def** **outliers\_check**(fig,columnnametocheck): # function with parameters given.

"""

The outlier\_check fuction plots outliers for the sucidedataframe.

The first parameter fig takes a number as argument which is the number of the figure bening plotted.

The second parameter columnnametocheck takes a column anme from sucidedataframe to plot its outliers.

"""

plt.figure(fig) # number of the figure being plotted.

plt.title("Outliers",fontsize=**18**) # Displays plot title

plt.boxplot(sucidedataframe[[columnnametocheck]]) # Displays description of the plots x and y labels.

plt.xlabel(columnnametocheck) # Displays the x axis for the plot

plt.ylabel(columnnametocheck) # Displays the y axis for the plot.

plt.grid() # adds a gird to the plot

plot\_values\_list = ["Population","Internet\_Usage\_per\_percent\_Population", "Total\_Labour\_force",\

"Cellular\_subscriptions\_per\_100\_people" , "GDPpyear", "GDPpcapital", "Expense\_percent\_of\_GDP)", "Physicians\_per\_1,000\_people)",\

"Refugee\_population\_by\_country\_or\_territory\_of\_origin", "Suicidesper100k"] # a list of column names in

# sucidedataframe to use for plotting.

**for** i **in** range(len(plot\_values\_list)): # looping in a range of the length of plot\_values\_list.

outliers\_check(i+**1**,plot\_values\_list[i]) # Arguemnts given to the function.

**def** **relationageandgender**(fig,columnnametocheck): # function with parameters given.

"""

The relationageandgender plots the relationship between a given column from the sucidedatafame with the

Age and Gender column form the same dataframe.

The first parameter fig takes a number as argument which is the number of the figure bening plotted.

The second parameter columnnametocheck takes a column name from sucidedataframe to check its relationship with the

Age and Gender column form the same dataframe.

"""

**print**("plot in respect to ", columnnametocheck) # printing the name of column to be plotted.

plt.figure(fig) # number of the figure being plotted.

plt.style.use('seaborn-white') # style to use for seaborn.

sns.catplot(x='Gender', y=columnnametocheck, col="Age", col\_wrap=**3**, sharey=True, \

data=sucidedataframe, alpha=**0.5**, palette = 'hot') # plots gender and Age in relationship with columnnametocheck.

plt.xticks(rotation = **90**) # rotates plot to 90 degree

plt.show() # shows plot.

**for** i **in** range(len(plot\_values\_list)): # looping in a range of the length of plot\_values\_list.

relationageandgender(i+**1**,plot\_values\_list[i]) # Arguemnts given to the function.

**def** **relationsucide100kwithage**(fig,columnnametocheck): # function with parameters given.

"""

The relationsucide100kwithage plots the relationship between a given column from the sucidedatafame with the

Suicidesper100k, Gender and Age column form the same dataframe.

The first parameter fig takes a number as argument which is the number of the figure bening plotted.

The second parameter columnnametocheck takes a column name from sucidedataframe to check its relationship with the

Suicidesper100k, Gender and Age column form the same dataframe.

"""

plt.figure(fig) # number of the figure being plotted.

My\_plot\_object = sns.FacetGrid(sucidedataframe , row = 'Gender',col = 'Age',margin\_titles=True)

My\_plot\_object.map(plt.scatter,"Suicidesper100k",columnnametocheck,edgecolor = 'w') # plots "Suicidesper100k" column in

# relation to columnnametocheck from sucidedataframe.

plt.show() # shows plot.

**for** i **in** range(len(plot\_values\_list)-**1**): # looping in a range of the length of plot\_values\_list.

relationsucide100kwithage(i+**1**,plot\_values\_list[i]) # Arguemnts given to the function.

# Function to perform OneHotEncoding

**def** **encodingoh**(required\_column,dropped\_column,column\_rename): # function to perform the one hot encoding with parameters given.

"""

encodingoh fuction perfoms OneHotEncoding on a given dataset column.

The first parameter required\_column is the column to be OneHotEncoded.

The second parameter dropped\_column checks if the is a column to be dropped when OneHotEncoding.

for example the arguemnt None drops no columns.

the argement "first" drops the fist column when OneHotEncoding.

The Third parameter column\_rename is the new names of the colums after OneHotEncoding

The encodingoh function returns a one hot encoded dataframe.

"""

oh = OneHotEncoder(drop=dropped\_column,dtype=np.int) # creates the instace of the object.

newdf = required\_column # creates a new data frame from the column to be one hot encoded

newdf = oh.fit\_transform(newdf).toarray() # one hot enocdes the new dataframe as a array

newdf = pd.DataFrame(newdf) # converts the newly created array to a dataframe.

newdf.columns = column\_rename # renames the newly encoded column

**print**(newdf.head(**5**)) # outputs the head of the dataframe.

**return** newdf # returns the one hot encoded data frame

### One Hot Encoding

sucidedataframe.nunique() # outputs unique values in each column in a data frame.

sucidedataframe.duplicated().sum() # gives the sum of duplicate dvalues

**print**(sucidedataframe.pivot\_table(columns=['Gender'], aggfunc='size')) # counts duplicates in the selected dataframe column.

**print**(sucidedataframe.pivot\_table(columns=['Country'], aggfunc='size')) # counts duplicates in the selected dataframe column.

sucidedataframe.head(**1**) # outputs the head of the dataframe

**print**(sucidedataframe.pivot\_table(columns=['Age'], aggfunc='size')) # counts duplicates in the selected dataframe column.

sucidedataframe.index = pd.RangeIndex(len(sucidedataframe.index)) # outputs indext of the data frame

sucidedataframe.head()

### One Hot Encoding

**print**(sucidedataframe.pivot\_table(columns=['Gender'], aggfunc='size')) # prints the unique values of the column.

**print**(sucidedataframe.pivot\_table(columns=['Age'], aggfunc='size')) # prints the unique values of the column.

# one hot enocdes the column using the created function.

age = encodingoh(sucidedataframe[["Age"]], None, ['15-24 years',"25-34 years", "35-54 years", \

"5-14 years", "55-74 years", "75+ years"])

# one hot enocdes the column using the created function.

gender = encodingoh(sucidedataframe[["Gender"]], "first", ["Gender"])

sucidedataframe = sucidedataframe.drop('Age', **1**) # column is dropped

sucidedataframe = sucidedataframe.drop('Gender', **1**) # column is dropped

sucidedataframe.head(**5**) # dataframe head is printed.

sucidedataframe = pd.concat([ gender,sucidedataframe],axis=**1**) # column is concatanated to dataframe

sucidedataframe.head() # fisrt elements of the dataframe are outptted

sucidedataframe = pd.concat([age,sucidedataframe],axis=**1**) # column is concatanated to dataframe

sucidedataframe # fisrt elements of the dataframe are outptted

**print**(sucidedataframe.pivot\_table(columns=['Gender'], aggfunc='size')) # prints the unique values of the column.

**print**(sucidedataframe.pivot\_table(columns=['15-24 years'], aggfunc='size')) # counts duplicates in the selected

# dataframe column.

**print**(sucidedataframe.pivot\_table(columns=["25-34 years"], aggfunc='size')) # counts duplicates in the selected

# dataframe column.

**print**(sucidedataframe.pivot\_table(columns=["35-54 years"], aggfunc='size')) # counts duplicates in the

# selected dataframe column.

**print**(sucidedataframe.pivot\_table(columns=["5-14 years"], aggfunc='size')) # counts duplicates in the

# selected dataframe column.

**print**(sucidedataframe.pivot\_table(columns=["55-74 years"], aggfunc='size')) # counts duplicates in the

# selected dataframe column.

**print**(sucidedataframe.pivot\_table(columns=["75+ years"], aggfunc='size')) # counts duplicates in the

# selected dataframe column.

sucidedataframe.head() # the first elements of the data frame are outputted

sucidedataframe.describe().round() # Shows the count, mean, std, min, 25%, 50%, 75% and

# max of a datframe.

# Checking and selecting higly correlated features

plt.figure(figsize = (**15**,**10**)) # sets the size of the matrix

correlation\_matrix = sucidedataframe.corr().round(**2**) # creates the correlation matrix

sns.heatmap(data = correlation\_matrix, annot = True) # shows correlation matrix

Features\_after\_corelation\_matrix = sucidedataframe[[ "Gender", "5-14 years", "15-24 years", "75+ years"]] # higher correlated

# features are selected

Features\_after\_corelation\_matrix.head() # first elements of the dataframe are outputted.

# count duplicates function learned from: https://datatofish.com/count-duplicates-pandas/

sucidedataframe.pivot\_table(columns=['Country'], aggfunc='size') # counts duplicates in the selected dataframe column.

each\_country = np.unique(sucidedataframe[["Country"]].values) # unique country rows are selected.

each\_country # array is outputted

Country = encodingoh(sucidedataframe[["Country"]], None, each\_country) # each country is one hot encoded at a time.2

sucidedataframe = pd.concat([Country,sucidedataframe],axis=**1**) # each new country is concatanated to original data frame.

sucidedataframe.head() # first elements of the data frame are out putted

Features\_after\_corelation\_matrix = pd.concat([Country,Features\_after\_corelation\_matrix],axis=**1**) # each new country is

# concatanated to highly correlate data frame.

Features\_after\_corelation\_matrix.head() # first elements of the data frame are out putted

**print**(sucidedataframe.columns) # columns of the data frame are printed

sucidedataframe = sucidedataframe.drop('Country', **1**) # country column is dropped.

**print**(sucidedataframe.columns) # columns of the data frame are printed

# Making lables

sucidedataframe.describe().round() # Shows the count, mean, std, min, 25%, 50%, 75% and

# max of a datframe.

# Selecting the values for the labels

**print**("max : ", **178** ) # prints the given input.

**print**("high : ", round(**178** / **2**)) # prints the given input.

**print**("medium :", round(**89** / **2**)) # prints the given input.

**print**("low : ", round(**44** / **2**)) # prints the given input.

**print**("min : ", round(**22** / **2**)) # prints the given input.

# Coverting the values to catagories

sucidedataframe.loc[(sucidedataframe["Suicidesper100k"] < **11**), "Suicidesper100k"] = **1** # Encoding values below 11 as 1.

sucidedataframe.loc[(sucidedataframe["Suicidesper100k"] >= **11**) & (sucidedataframe["Suicidesper100k"] < **22**),\

"Suicidesper100k"] = **2**

# Encoding values aboveor equal to 11 as and below 22 to 2.

sucidedataframe.loc[(sucidedataframe["Suicidesper100k"] >= **22**) & (sucidedataframe["Suicidesper100k"] < **44**),\

"Suicidesper100k"] = **3**

# Encoding values equal to and above 22 and below 44 to 3.

sucidedataframe.loc[(sucidedataframe["Suicidesper100k"] >= **44**) & (sucidedataframe["Suicidesper100k"] < **89**),\

"Suicidesper100k"] = **4**

# Encoding values above or equal to 44 as and below 89 to 4.

sucidedataframe.loc[(sucidedataframe["Suicidesper100k"] >= **89**) & (sucidedataframe["Suicidesper100k"] <= **178**), \

"Suicidesper100k"] = **5**

# Encoding values above or equal to 89 as and below 178 to 4.

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

# Label Encoding

le = LabelEncoder() # creating the instance of the object.

sucidedataframe.Suicidesper100k = le.fit\_transform(sucidedataframe.Suicidesper100k) # label encoing the require dcolumn.

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

**print**(sucidedataframe.pivot\_table(columns=['Suicidesper100k'], aggfunc='size')) # counts duplicates in the selected

# dataframe column.

**9264** + **2850** + **2177** + **678** + **141** # calculation

sucidedataframe.head() # outputting dataframe head.

one = str(round(**100** \* (**9264**/**15110**),**2**)) + "%" # percentage of the value is checked from label column.

two = str(round(**100** \* (**2850**/**15110**),**2**)) + "%" # percentage of the value is checked from label column.

three = str(round(**100** \* (**2177**/**15110**),**2**)) + "%" # percentage of the value is checked from label column.

four = str(round(**100** \* (**678**/**15110**),**2**)) + "%" # percentage of the value is checked from label column.

five = str(round(**100** \* (**141**/**15110**),**2**)) + "%" # percentage of the value is checked from label column.

**print**("1 is represented ", one, "**\n**2 is represented ", two, "**\n**3 is represented ", three, "**\n**4 is represented ", four, "**\n**5 is represented ", five)

# percentage of the value is printed of label column.

plt.bar([**1**,**2**,**3**,**4**,**5**],[**9262**, **2850**, **2177**, **678**, **141**]) # bar chart is outputted to show the difference in values.

sucidedataframe.head() # dataframe head is outputted.

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

### selecting X and Y values

sucidedataframe.head(**1**) # the fisrt elements of the data frame are outputted

Features\_after\_corelation\_matrix.head(**1**) # the fisrt elements of the data frame are outputted

X = sucidedataframe.iloc[:, :-**1**].values # selecting the values for the X variable.

X2 = Features\_after\_corelation\_matrix.iloc[:, :].values # selecting the

# values for the X2 variable.

y = sucidedataframe.iloc[:, -**1**].values # selecting the values for the Y variable. # done using .to\_numpy and not

# .iloc as .to\_numpy creates a horizontal bar while .iloc creates a

# horizontal bar which will not alighn with the x values.

**print**("X ", X, "**\n**", "X2 " , X2 , "**\n**", "y ", y) # priting arrays

# Function to check principal Components

**def** **pcafuntion**(X,pcomponents ,pcolumns, y\_value, y\_var, plot\_labels): # A function to perfrom PCA with paramenters given.

"""

The pcafunction gives the principal components of a given datafame with respect to the selected label.

It also plots a 2d and 3d PCA plot if the number of pca are 2 or 3 respectively.

The first parameters X is the columns from a dataframe as a array to be converted to principal components.

The second parameter pcomponents is the name of the principal componets that are to be added as a list.

For example ['principal component 1','principal component 2'].

The third parameter y\_value is the column from the dataframe to be used as the y variable.

For example df[["y\_var"]]

The fourth parameter y\_var is the name of the y column for the dataframe.

The fifth parameter plot\_labels is the legends for the plot.

The pcafunction function returns the given dataframes required principal comonents concanated with the

y label as a dataframe.

"""

# Normalizing data before applying pca

sc = StandardScaler() # creating an instance of the object.

X = sc.fit\_transform(X) # selecting vaues to perfom transformation on

pca = PCA(n\_components=pcomponents) # selecting number of principal components

principalComponents = pca.fit\_transform(X) # # selecting values to perfom transformation on

principalDf = pd.DataFrame(data = principalComponents # creating a new dataframe with principal components.

, columns =pcolumns)

finalDf = pd.concat([principalDf, y\_value], axis = **1**) # concnating principal componenets with Class.

**print**(finalDf) # looking at the out put from pca.

**print**("**\n**",pca.explained\_variance\_ratio\_)# shows how much information is reatined from the pca

missing = **0** # setting variable to 0.

**for** i **in** range(len(pca.explained\_variance\_ratio\_)): # for looping in length of the array pca.explained\_variance\_ratio\_

missing = missing + pca.explained\_variance\_ratio\_[i] # adding pca.explained\_variance\_ratio\_[i]

# varables values to missing variable

**print**("Total values : ", missing, "%") # total data left

missing = **1** - missing # subtracting missing variables value from 1.

**print**("missing values :", missing, "%") # total data lost

**if** pcomponents == **2**: # checks the number of principal components.

plot = sns.relplot(

x='principal component 1', # selecting first princal component.

y='principal component 2', # selecting second princal component.

hue=y\_var, # sets hue to the y variable.

data=finalDf, # selects where to get data from.

facet\_kws={'legend\_out': False} # helps in selecting a custom legend.

) # plots a scatter plot of principal components.

plt.title('Dataset with pca') # Title of the plot

# check axes and find which is have legend

leg = plot.axes.flat[**0**].get\_legend()

new\_title =y\_var # Legend title.

leg.set\_title(new\_title) # setting title to legend.

leg.get\_frame().set\_alpha(**255**) # setting transparency level for legend box

labels = plot\_labels # label list for legend

**for** t, l **in** zip(plot.\_legend.texts, labels): # looping through a touple.

t.set\_text(l) # adding label to list

**elif** pcomponents == **3**: # checks the number of principal components.

fig = plt.figure(figsize=(**15**,**10**)) # setting plot size

ax = plt.axes(projection='3d') # specifying that it is a 3d plot.

plt.title('Dataset with pca', fontsize=**20**) # adding a title to the plot

# Data for three-dimensional scattered points

xdata = finalDf['principal component 1'] # selecting first princal component.

ydata = finalDf['principal component 2'] # selecting second princal component.

zdata = finalDf['principal component 3'] # selecting third princal component.

ax.scatter3D(xdata, ydata, zdata, c=y\_value) # plotting princhpla components

ax.set\_xlabel("First Principal Component",fontsize=**15**) # setting label for first principal componenet.

ax.set\_ylabel("Second Principal Component",fontsize=**15**) # setting label for second principal componenet.

ax.set\_zlabel("Third Principal Component",fontsize=**15**) # setting label for third principal componenet.

plt.show() # making plot visible.

**return** finalDf # returning finalDf

# Principal component analysis

## Using continues variables

X\_pca = sucidedataframe.iloc[:, **55**:].values

pcafuntion(X\_pca,**2** ,['principal component 1','principal component 2'], sucidedataframe[["Suicidesper100k"]], \

"Suicidesper100k", [**1**,**2**,**3**,**4**,**5**])

pcafuntion(X\_pca,**3** ,['principal component 1','principal component 2', 'principal component 3'], \

sucidedataframe[["Suicidesper100k"]], "Suicidesper100k", [**1**,**2**,**3**,**4**,**5**])

pcom = "principal component "

plist = ['principal component 1','principal component 2', 'principal component 3']

**for** i **in** range(**4**,**10**):

plist.append(pcom + str(i))

**print**("Number of Components ", i)

pcafuntion(X\_pca,i ,plist, sucidedataframe[["Suicidesper100k"]], "Suicidesper100k", [**1**,**2**,**3**,**4**,**5**])

**print**("**\n\n**")

## Over-sampling

**print**('Original dataset shape %s' % Counter(y)) # original data set rows counted

ros = RandomOverSampler(random\_state=**42**) # instance of object created.

X\_over, y\_over = ros.fit\_resample(X, y) # oversampling model fitting

**print**('Resampled dataset shape %s' % Counter(y\_over)) # oversampled data set rows counted

## Spliting the datasets into a training and test set

## Original x and y values.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = **0.2**, random\_state = **1**)

# test\_size = 0.2 # splitting the data into 80 and 20 percent between the training and test set

# to get the best results.

# random\_state = 1 # resetting the random seed

# print the lenghth of both test and train set to see if there equally split.

**print**("The length of X\_train is ",len(X\_train), " and the length of y\_train is ", len(y\_train))

**print**("The length of X\_test is ",len(X\_test), " and the length of y\_test is ", len(y\_test))

## Original x and y values after they have been over Sampled.

X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over = train\_test\_split(X\_over, y\_over, test\_size = **0.2**, random\_state = **1**)

# test\_size = 0.2 # splitting the data into 80 and 20 percent between the training and test set

# to get the best results.

# random\_state = 1 # resetting the random seed

# print the lenghth of both test and train set to see if there equally split.

**print**("The length of X\_train\_over is ",len(X\_train\_over), " and the length of y\_train\_over is ", len(y\_train\_over))

**print**("The length of X\_test\_over is ",len(X\_test\_over), " and the length of y\_test\_over is ", len(y\_test\_over))

## x2 and y2 values choosend by using the corelation matrix.

X2\_train, X2\_test, y2\_train, y2\_test = train\_test\_split(X2, y, test\_size = **0.2**, random\_state = **1**)

# test\_size = 0.2 # splitting the data into 80 and 20 percent between the training and test set

# to get the best results.

# random\_state = 1 # resetting the random seed

# print the lenghth of both test and train set to see if there equally split.

**print**("The length of X2\_train is ",len(X2\_train), " and the length of y2\_train is ", len(y2\_train))

**print**("The length of X2\_test is ",len(X2\_test), " and the length of y2\_test is ", len(y2\_test))

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

# Different models are evaluated.

# Function to evaluate different models

**def** **Classifier\_function**(model, X\_train, y\_train,X\_test,y\_test, title): # function takes the name of the

"""

The Classifier\_function Checks, predicted/actual results , checks testing and traning scores,

Checks Actual values classified correctly and wrongly Checks accuracy, precision, recall, f1

scores and area under the curve.

It plots a confusion matix with accuracy, precision, recall, f1 scores given.

The first parameter is model the model to evaluate.

The secound parameter X\_train is the variable contaning training datas features.

The third parameter y\_train is the variable contaning training datas label.

The fourth parameter X\_test is the variable contaning testing datas features.

The fifth parameter y\_test is the variable contaning testing datas label.

The sixth parameter is the string which will be used as the title for the confusion matrix.

The Classifier\_function function returns the train\_accuracy,test\_accuracy , precision, recall,

f1\_score and Area\_under\_the\_curve of a given model.

"""

# model used, the x and y traning and testing sets.

model.fit(X\_train, y\_train) # Building the k-nearest neighbors classification model.

y\_test\_p = model.predict(X\_test) # Predicted results.

**print**(" results**\n**pred-Actual") # printing predicted and real values.

**print**(np.concatenate((y\_test\_p.reshape(len(y\_test\_p),**1**),y\_test.reshape(len(y\_test),**1**)),**1**)) # Predicted results and

# real results in a np array.

train\_accuracy = round(model.score(X\_train,y\_train),**2**) \* **100** # Getting traing accuracy multipling it by 100 after

# rounding it by 2 to get a score between 0 to 100

test\_accuracy = round(model.score(X\_test,y\_test),**2**) \* **100** # Getting testing accuracy multipling it by 100 after

# rounding it by 2 to get a score between 0 to 100

**print**("Model train accuracy: ", train\_accuracy, "%") # printing the model accurcy.

**print**("Model test accuracy: ", test\_accuracy, "%") # printing the model accurcy.

**print**("**\n\n**") # printing a new line.

# getting Accuracy or recall or precision or specificity

y\_test\_pred = model.predict(X\_test) # predicted results

cReport = classification\_report(y\_test,y\_test\_pred) # creating a Classification report

**print**(cReport) # creating a Classification report

cm = confusion\_matrix(y\_test, y\_test\_p) # creating the confusion matrix

cm2 = multilabel\_confusion\_matrix(y\_test, y\_test\_pred) # creating a mutable confusion matrix

precision, recall, f1\_score, support = precision\_recall\_fscore\_support(y\_test, y\_test\_pred) # getting the precision,

# recall and f1score for later use.

accuracy = round(np.trace(cm) / float(np.sum(cm)), **2**) \* **100** # getting aaccuracy and multipling it by 100 after

# rounding it by 2 to get a score between 0 to 100.

precision = round(np.mean(precision),**2**) \* **100** # multipling precision variables mean by 100 after rounding it by

# 2 to get a score between 0 to 100.

recall = round(np.mean(recall),**2**) \* **100** # multipling recall variables mean by 100 after rounding it by 2 to

# get a score between 0 to 100.

f1\_score = round(np.mean(f1\_score),**2**) \* **100** # multipling f1\_score variables mean by 100 after rounding

# it by 2 to get a score between 0 to 100.

lable\_list = [] # creating a empty list

**for** i **in** range(len(cm)): # looping in the range of the length of the confusion matrix.

**for** j **in** range(len(cm)): # looping in the range of the length of the confusion matrix.

**if** j == i: # if the value of j is equal to the value of i.

# the below code appends the Actual Values Classified correctly to the variable lable\_list.

lable\_list.append("Actual "+ str(i) +"**\n**" + "calssified as "+ str(j) +"**\n**" + str(cm[i][j]) + "**\n**"+ \

str(round(cm[i][j]/np.sum(cm),**2**)) + " %")

**else**: # otherwise

# the below function appends the actual values classified wrongly to the variable lable list.

lable\_list.append("Actual "+ str(i) +"**\n**" + "calssified as "+ str(j) + "**\n**" + str(cm[i][j]) + "**\n**"+ \

str(round(cm[i][j]/np.sum(cm),**2**)) + " %")

lable\_list = np.asarray(lable\_list).reshape(len(cm),len(cm)) # resahping the label list as a numpy array to be

# used in plotting the confusion matrix

# the variable function will be will be used to display the results of the evaluation to the confusion matrix.

total\_score = ("Accuracy: " + str(accuracy) +" %" + "**\n**Precison: " + str(precision) +" %" + "**\n**Recall: " +\

str(recall) +" %" + "**\n**F1 score: " + str(f1\_score) +" %")

# Below is the code used to plot the confusion matrix.

plt.figure(figsize = (**12**,**9**)) # sets the size of the matrix

disp = sns.heatmap(cm, annot=lable\_list, fmt='', cmap='Blues') # displays the results of the actual values

# classified wrongly and correctly.

disp.plot() # displaying data in plot

plt.title(title, fontsize=**25**) # adding a title to plot

plt.ylabel('True label', fontsize=**20**) # adding a y axis to the plot.

plt.xlabel('Predicted label' +"**\n\n**Scores**\n**" +total\_score, fontsize=**20**) # adding a x axis to the plot

plt.show() # showing the plot

y\_pred\_proba = model.predict\_proba(X\_test)

Area\_under\_the\_curve = roc\_auc\_score(y\_test, y\_pred\_proba, multi\_class="ovr")

**print**("Area under the curve: ", Area\_under\_the\_curve)

result\_list = train\_accuracy,test\_accuracy , precision, recall, f1\_score,Area\_under\_the\_curve # returning the results

**return** result\_list # returns the results from the model.

# Feature Scaling and testing function

**def** **featurescaling**(Scaler, X\_train, X\_test, y\_train, y\_test, Modelandprams ,Modelname):

"""

The featurescaling fundtion feature scales given features and then Checks, predicted/actual results , checks testing and traning scores,

Checks Actual values classified correctly and wrongly Checks accuracy, precision, recall, f1

scores and area under the curve.

It plots a confusion matix with accuracy, precision, recall, f1 scores given.

The first parameter is typpe of scaling technique to use.

The secound parameter X\_train is the variable contaning training datas features.

The third parameter y\_train is the variable contaning training datas label.

The fourth parameter X\_test is the variable contaning testing datas features.

The fifth parameter y\_test is the variable contaning testing datas label.

The sixth parameter is parameter is model the model to evaluate.

The seventh parameter is the string which will be used as the title for the confusion matrix.

The featurescaling function returns the train\_accuracy,test\_accuracy , precision, recall,

f1\_score and Area\_under\_the\_curve of a given model after feature scaling.

"""

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

sc = Scaler # creating an instance of the object.

**print**("Before scaling:**\n**X\_test ", X\_test,"**\n\n**X\_train ", X\_train) # printing the sets before feature scaling.

X\_train[:, **55**:] = sc.fit\_transform(X\_train[:, **55**:]) # Scaling x\_train

X\_test[:,**55**:] = sc.transform(X\_test[:, **55**:]) # Scaling y\_train

**print**("After scaling:**\n**X\_test ", X\_test,"**\n\n**X\_train ", X\_train) # printing the sets after feature scaling.

**print**("**\n**The result of the model")

train\_accuracy,test\_accuracy , precision, recall, f1\_score, Area\_under\_the\_curve =Classifier\_function(Modelandprams ,\

X\_train, y\_train,X\_test, y\_test, Modelname)

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

**return** train\_accuracy,test\_accuracy , precision, recall, f1\_score, Area\_under\_the\_curve

# Evaluating different models to choose which one to use

## k-fold cross-validation

**def** **crossvalscore**(model, X, y, cv\_val): # function to perform cross validation with model X, y and cv\_val as parameters

"""

The crossvalscore function prints the avarage coross validation accuracy of a model and its standard deveation.

The first parameter model is the type of model to perform the cross validation on.

The second parameter X is the features.

The third parameter y is the labels.

the foruth parameter cv\_val is the number of times to cross validate the given model.

The crossvalscore function returns the mean accuracy of the model.

"""

score = cross\_val\_score(estimator = model, X = X, y = y, cv = cv\_val) # performs different tests to get best accurecy.

**print**("Accuracy: {:.2f} %".format(score.mean()\***100**)) # accuracy printed.

**print**("Standard Deviation: {:.2f} %".format(score.std()\***100**)) # standard deveation printed (std -avarage or std+ avarage )

**return** round(score.mean(),**2**)\***100**, round(score.std(),**2**)\***100**

model\_name = ["Logistc Regression", "KNN", "Naive Bayes", "Neural Networks", "Decision Tree", "Random Forest"]

Accuracy\_list = [] # making empty list

std\_list = [] # making empty list

# Logistic

crossvalscore(LogisticRegression(), X, y, **10**) # Performs cross validation.

Accuracy, std = crossvalscore(LogisticRegression(), X\_over, y\_over, **10**) # Performs cross validation.

Accuracy\_list.append(Accuracy) # appending to list

std\_list.append(std) # appending to list

crossvalscore(LogisticRegression(), X2, y, **10**) # Performs cross validation.

# KNN

crossvalscore(KNeighborsClassifier(), X, y, **10**) # Performs cross validation.

Accuracy, std = crossvalscore(KNeighborsClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

Accuracy\_list.append(Accuracy) # appending to list

std\_list.append(std) # appending to list

crossvalscore(KNeighborsClassifier(), X2, y, **10**) # Performs cross validation.

# Naive bayes

crossvalscore(GaussianNB(), X, y, **10**) # Performs cross validation.

Accuracy, std = crossvalscore(GaussianNB(), X\_over, y\_over, **10**) # Performs cross validation.

Accuracy\_list.append(Accuracy) # appending to list

std\_list.append(std) # appending to list

crossvalscore(GaussianNB(), X2, y, **10**) # Performs cross validation.

# Neural Network

crossvalscore(MLPClassifier(), X, y, **10**) # Performs cross validation.

Accuracy, std = crossvalscore(MLPClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

Accuracy\_list.append(Accuracy) # appending to list

std\_list.append(std) # appending to list

crossvalscore(MLPClassifier(), X2, y, **10**) # Performs cross validation.

# Decision Tree

crossvalscore(DecisionTreeClassifier(), X, y, **10**) # Performs cross validation.

Accuracy, std = crossvalscore(DecisionTreeClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

Accuracy\_list.append(Accuracy) # appending to list

std\_list.append(std) # appending to list

crossvalscore(DecisionTreeClassifier(), X2, y, **10**) # Performs cross validation.

# Random Forest

crossvalscore(RandomForestClassifier(), X, y, **10**) # Performs cross validation.

Accuracy, std = crossvalscore(RandomForestClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

Accuracy\_list.append(Accuracy) # appending to list

std\_list.append(std) # appending to list

crossvalscore(RandomForestClassifier(), X2, y, **10**) # Performs cross validation.

Evaluation2 = pd.DataFrame({

'Model':model\_name,

'Crossvaladation Accuracy':Accuracy\_list,

'Standard deveation':std\_list,

}) # using lists to make a dataframe

Evaluation2

Evaluation2.sort\_values(by='Crossvaladation Accuracy', ascending=False) # viewing dataframe information by recall in decending order.

Evaluation2.sort\_values(by='Standard deveation', ascending=True) # viewing dataframe information by recall in decending order.

**def** **plot\_bar\_graph**(list\_for\_plot, list\_for\_plot2,Accuracy\_type, colour): # funstion with given parameters.

"""

The function plot\_bar\_graph plots a bar graph.

The first parameter list\_for\_plot is the x axis.

The second parameter list\_for\_plot2 is the y axis.

The third parameter Accuracy\_type takes a string.

for example "F1 score"

The fourth parameter colour is the color for the bar plot.

"""

plt.figure() # plots figure

plt.title("Model Evaluation",fontsize=**18**) # Displays plot title

plt.bar(list\_for\_plot, list\_for\_plot2, color = colour) # Displays description of the plots x and y labels.

plt.xlabel("Models") # Displays the x axis for the plot

plt.ylabel(Accuracy\_type) # Displays the y axis for the plot.

plt.grid() # adds a gird to the plot

plot\_bar\_graph(model\_name, Accuracy\_list , "accuracy", "red") # functions argents given. plots output using given data.

plot\_bar\_graph(model\_name, std\_list , "standard deveation", "blue") # functions argents given. plots output using given data.

# KNN Model

## k-fold cross-validation Graph

**def** **subplot**(list\_for\_plot, ave\_scores, hyper\_pram\_to\_use\_name): # A function with parameters given.

"""

The function subplot plots a linear plot and shows the best results using two lists

the fist one acting as a label to the second list and the second list being scores.

The first parameter list\_for\_plot is the X label.

The second parameter ave\_scores is the y label.

The thrid paramater hyper\_pram\_to\_use\_name is the the titile for the plot.

"""

**print**(hyper\_pram\_to\_use\_name + " ", list\_for\_plot,"**\n**Average " + hyper\_pram\_to\_use\_name +\

" scores", ave\_scores) # printing the average score of the model

**for** i **in** range(len(ave\_scores)): # looping to the legth of list

**if** ave\_scores[i] == max(ave\_scores): # if the first value is equal to the second value

**print**("The hyperparameter",list\_for\_plot[i], "gives the best result :", ave\_scores[i]) # print output.

**print**("**\n\n**") # print new line

plt.figure() # plot a figure

plt.title("Best " + hyper\_pram\_to\_use\_name + " Selection",fontsize=**18**) # Displays plot title

plt.plot(list\_for\_plot, ave\_scores) # Displays description of the plots x and y labels.

plt.xlabel(hyper\_pram\_to\_use\_name +" values") # Displays the x axis for the plot

plt.ylabel("Average CV model accuracy") # Displays the y axis for the plot

plt.legend([hyper\_pram\_to\_use\_name], loc="lower right") # adds a legend to the plot.

plt.grid() # adds a gird to the plot

**def** **K\_NN\_plot\_Values**(X,y): # A function with parameters given.

"""

The K\_NN\_plot\_Values performs cross valadation on the KNN models all hyperparametes, prints

the best hyperparameters that give the best reults and plots the results

The first parameter X takes the features.

The first parameter y takes the lables.

"""

ave\_scores = [] # Creating a empty list

k\_list = k\_list = list(range(**1**,**50**)) # Setting the values for the list variable.

**for** single\_val **in** k\_list: # looping through each value in the k\_list variable

model = KNeighborsClassifier(n\_neighbors = single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**,scoring="accuracy") # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(k\_list, ave\_scores, "n\_neighbors") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

weight\_list = ["uniform", "distance"] # weights list uniform and distance

**for** single\_val **in** weight\_list: # looping through each value in the list variable

model = KNeighborsClassifier(weights = single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**,scoring="accuracy") # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(weight\_list, ave\_scores, "weights") # using the function to plot x and y values

ave\_scores = [] # Creating a empty list

metric\_list = ['minkowski','euclidean','manhattan'] # a list of KNN metrics

**for** single\_val **in** metric\_list: # looping through each value in the list variable

model = KNeighborsClassifier(metric = single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**,scoring="accuracy") # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(metric\_list, ave\_scores, "metric") # using the function to plot x and y values

ave\_scores = [] # Creating a empty list

algorithm\_list = ['auto', 'ball\_tree', 'kd\_tree', 'brute']

**for** single\_val **in** algorithm\_list: # looping through each value in the list variable

model = KNeighborsClassifier(algorithm = single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**,scoring="accuracy") # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(algorithm\_list, ave\_scores, "algorithm") # using the function to plot x and y values

ave\_scores = [] # Creating a empty list

leaf\_size\_list = list(range(**1**,**30**)) # a list of KNN metrics

**for** single\_val **in** leaf\_size\_list: # looping through each value in the list variable

model = KNeighborsClassifier(algorithm = 'ball\_tree', leaf\_size = single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**,scoring="accuracy") # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(leaf\_size\_list, ave\_scores, "Algorithm Ball tree leaf\_size") # using the function to plot x and y values

ave\_scores = [] # Creating a empty list

leaf\_size\_list = list(range(**1**,**30**)) # a list of KNN metrics

**for** single\_val **in** leaf\_size\_list: # looping through each value in the list variable

model = KNeighborsClassifier(algorithm = 'kd\_tree', leaf\_size = single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**,scoring="accuracy") # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(leaf\_size\_list, ave\_scores, "Algorithm Kd leaf\_size") # using the function to plot x and y values

# X and y un\_edited

## Plotting the KNN's with a average Cross-val score

K\_NN\_plot\_Values(X,y) # PLots KNN Cross Val score

# Over Sampled X and y values

## Plotting the KNN's with a average Cross-val score

K\_NN\_plot\_Values(X\_over,y\_over) # PLots KNN Cross Val score

# X and y obtained by using Corelation matrix

## Plotting the KNN's with a average Cross-val score

K\_NN\_plot\_Values(X2,y) # PLots KNN Cross Val score

# X and y un\_edited

# Without any model Tuning

crossvalscore(KNeighborsClassifier(), X, y, **10**) # Performs cross validation.

# Using Cross Validation hyperparameter selection

crossvalscore(KNeighborsClassifier(weights = 'uniform', n\_neighbors = **30**), X, y, **10**) # Performs cross validation.

# Over Sampled X and y values

# Without any model Tuning

crossvalscore(KNeighborsClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

# Using Cross Validation hyperparameter selection

crossvalscore(KNeighborsClassifier(weights = 'distance', n\_neighbors = **1**), X\_over, y\_over, **10**) # Performs cross validation.

# X and y obtained by using Corelation matrix

# Without any model Tuning

crossvalscore(KNeighborsClassifier(), X, y, **10**) # Performs cross validation.

# Using Cross Validation hyperparameter selection

leaf\_size\_list = [**1**,**24**,**25**,**26**,**27**,**28**,**29**] # a list of leaf sizes.

**for** lsz **in** leaf\_size\_list: # looping in a list.

**print**("When Leaf Size = ", lsz) # printing output

crossvalscore(KNeighborsClassifier(weights = 'uniform', n\_neighbors = **17**,algorithm = 'ball\_tree', \

leaf\_size = lsz), X, y, **10**) # Performs cross validation.

**print**("**\n**") # printing a new line.

leaf\_size\_list = [**1**,**24**,**25**,**26**,**27**,**28**,**29**] # a list of leaf\_size

**for** lsz **in** leaf\_size\_list: # looping through a list

**print**("When Leaf Size = ", lsz) # printing output

crossvalscore(KNeighborsClassifier(weights = 'distance', n\_neighbors = **17**,algorithm = 'ball\_tree', \

leaf\_size = lsz), X, y, **10**) # Performs cross validation.

**print**("**\n**") # printing new line

leaf\_size\_list = [**24**,**25**,**26**,**27**,**28**,**29**] # a list of leaf\_size

**for** lsz **in** leaf\_size\_list: # looping through a list

**print**("When Leaf Size = ", lsz) # printing output

crossvalscore(KNeighborsClassifier(weights = 'uniform', n\_neighbors = **17**,algorithm = 'ball\_tree', \

leaf\_size = lsz), X, y, **10**) # Performs cross validation.

**print**("**\n**") # printing new line

leaf\_size\_list = [**24**,**25**,**26**,**27**,**28**,**29**]

**for** lsz **in** leaf\_size\_list:

**print**("When Leaf Size = ", lsz)

crossvalscore(KNeighborsClassifier(weights = 'distance', n\_neighbors = **17**,algorithm = 'ball\_tree', \

leaf\_size = lsz), X, y, **10**) # Performs cross validation.

**print**("**\n**")

# Random Search plot

**def** **Random\_search\_plot**(sorted\_list1, sorted\_list2, sorted\_uniform, sorted\_distance, single\_val):

"""

The function Random\_search\_plot is used to make a plot for the random search out put for KNN.

The first parameter sorted\_list1 is a list used for the first x axis.

The secound parameter sorted\_list2 is a list used for the secound x axis.

The third parameter sorted\_uniform is a list used for the first y axis.

The fourth parameter sorted\_distance is a list used for the second y axis.

The fifth parameter single\_val is used as the title of the plot

"""

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Random Search Hyper parameter Comparison with " + single\_val + " metric",\

fontsize=**25**, color="blue") # setting the plot title.

plt.plot(sorted\_list1,sorted\_uniform , "--",marker = "o", label="weights uniform", color="red") # setting the

# uniform variables

# values to the plot on the y axis and K\_list on the x axis.

plt.plot(sorted\_list2,sorted\_distance , ":",marker = "\*", label="weights distant", color="blue") # setting the distant

# variables values to the plot on the y axis and K\_list

# on the x axis.

plt.xlabel("Number of nearest neighbours", fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score", fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# sub function to preform Random Search

**def** **rand\_search\_fun**(typeofmodelandprams, dict\_prams,crossval, X, y): # Function takes in the model type,

# number of crossvalidation sand X and y values as hyperparameters.

"""

The function rand\_search\_fun performs random search on a model. It prints the best results and

the hyperparametrs used to obtain those results and then it plots those results.

The first parameter typeofmodelandprams is the type of model used on which random search is performed on.

The secound parameter dict\_prams is a dictinory of hyperparameters to be used in the random search.

The third parameter crossval is the number of cross validations to be performed.

The fourth parameter X is the features.

The fifth parameter y is the labels.

The function rand\_search\_fun returns the results of the random search.

"""

model = typeofmodelandprams # creating an instance of the object.

parameters = [dict\_prams] # hyper parameters for the random search

rand\_search = RandomizedSearchCV(

model,

#estimator = model, # model

#param\_distributions = parameters, # hyper paramaters

parameters,

scoring = 'accuracy', # score measurement

cv = crossval, # number of cross validations

n\_jobs = -**1**, # selecting all possible paramaters to go through to get the best model possible

return\_train\_score=False, # train score is false as it can be computationaly

# expensive. without storing the traning score the grd search is fater

n\_iter=**10**, # setting the number of iterations

random\_state=**5**)

rand\_search.fit(X, y) # applying the search on our model.

#print(pd.DataFrame(rand\_search.cv\_results\_)[["mean\_test\_score","params"]])

**print**(pd.DataFrame(rand\_search.cv\_results\_)) # to print the whole result

best\_accuracy = rand\_search.best\_score\_ # the best accuracy

best\_parameters = rand\_search.best\_params\_ # the best paramaters that gave the best accurecy

**print**("Best Accuracy: {:.2f} %".format(best\_accuracy\***100**)) # printing best accuracy

**print**("Best Parameters:", best\_parameters) # printing the best parameters

**print**(parameters) # prininting the parameters

**return** rand\_search # return random search value.

# Performs full Random Search on KNN Model

**def** **KNN\_full\_rand\_search**(X, y): # performs randomized search

"""

The function KNN\_full\_rand\_search performs random search on KNN model. It prints the best results and

the hyperparametrs used to obtain those results and then it plots those results.

The first parameter X is the features.

The secound parameter y is the labels.

"""

metric\_list = ['minkowski','euclidean','manhattan']

k\_list = list(range(**1**,**31**)) # list from 1 - 30 (will be used as KNN's)

**for** single\_val **in** metric\_list:

weight\_list = ["uniform"] # weights list uniform and distance

para\_rand = dict(n\_neighbors=k\_list,weights=weight\_list, metric=[single\_val]) # adding the above lists in a dictinorr.

rand\_search = rand\_search\_fun(KNeighborsClassifier(), para\_rand, **5**, X, y) # Using the grid search gunction with

# arguments given

uniform = [] # creating a empty list

k\_list1 = [] # creating a empty list

**for** i **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times the length of

# the mean\_test\_score

uniform.append(rand\_search.cv\_results\_["mean\_test\_score"][i]) # appending the uniform values ot the

# uniform variable.

k\_list1.append(rand\_search.cv\_results\_["param\_n\_neighbors"][i]) # appedning the KNN's to the K\_list1.

sorted\_uniform = [] # creating a empty list

sorted\_list1 = k\_list1.copy() # creating a copy of the K\_list1 variables values.

sorted\_list1.sort() # sorting the KNNs

**for** i **in** range(len(sorted\_list1)): # Looping through the lenght of the sorted\_list1 variable.

**for** j **in** range(len(k\_list1)): # Looping through the lenght of the K\_list1 variable.

**if** sorted\_list1[i] == k\_list1[j]: # checking that if the values of sorted\_list1[i] is equal to k\_list1[j]

sorted\_uniform.append(uniform[j]) # appeding the unifrom[j] values to the sorted\_uniform list.

weight\_list = ["distance"] # weights list distance

para\_rand = dict(n\_neighbors=k\_list,weights=weight\_list,metric=[single\_val]) # adding the above lists in a dictinory.

**print**("**\n**unsorted kNN ",k\_list1) # checking out put of the given variables values

**print**("sorted KNN ", sorted\_list1) # checking out put of the given variables values

**print**("unsorted Uniform ", uniform) # checking out put of the given variables values

**print**("sorted uniform ", sorted\_uniform, "**\n**") # checking out put of the given variables values

rand\_search = rand\_search\_fun(KNeighborsClassifier(), para\_rand, **5**, X, y) # Using the grid search gunction with

# arguments given

distance = [] # creating a empty list

k\_list2 = [] # creating a empty list

**for** i **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times the length of

# the mean\_test\_score

distance.append(rand\_search.cv\_results\_["mean\_test\_score"][i]) # appending the distance values ot the

# distance variable.

k\_list2.append(rand\_search.cv\_results\_["param\_n\_neighbors"][i]) # appedning the KNN's to the K\_list2.

sorted\_distance = [] # creating a empty list

sorted\_list2 = k\_list2.copy() # creating a copy of the K\_list2 variables values.

sorted\_list2.sort() # sorting the KNNs

**for** i **in** range(len(sorted\_list2)): # Looping through the lenght of the sorted\_list2 variable.

**for** j **in** range(len(k\_list2)): # Looping through the lenght of the K\_list2 variable.

**if** sorted\_list2[i] == k\_list2[j]: # checking that if the values of sorted\_list2[i] equal k\_list2[j]

sorted\_distance.append(distance[j]) # appeding the unifrom[j] values to the sorted\_uniform list.

**print**("**\n**unsorted KNN ", k\_list2) # checking out put of the given variables values

**print**("sorted KNN ", sorted\_list2) # checking out put of the given variables values

**print**("unsorted distance ", distance) # checking out put of the given variables values

**print**("sorted distance ", sorted\_distance,"**\n**") # checking out put of the given variables values

Random\_search\_plot(sorted\_list1, sorted\_list2, sorted\_uniform, sorted\_distance, single\_val)

# X and y un\_edited

## Random search

KNN\_full\_rand\_search(X, y) # performs random search.

# Over Sampled X and y values

## Random search

KNN\_full\_rand\_search(X\_over, y\_over) # performs random search.

# X and y obtained by using Corelation matrix

## Random search

KNN\_full\_rand\_search(X2, y) # performs random search.

# Grid Search plot

**def** **grid\_search\_plot**(distance, uniform, k\_list, single\_val):

"""

The function grid\_search\_plot is used to make a plot for the grid search out put for KNN.

The first parameter distance is a list used for the second y axis.

The secound parameter uniform is a list used for the first y axis.

The third parameter k\_list is a list used for the first x and second axis.

The fourth parameter single\_val is used as the title of the plot

"""

**print**("distance ", distance) # printing the values distace variable.

**print**("uniform ", uniform) # printing the values uniform variable.

**print**("**\n\n\n**")

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Grid Search Hyper parameter Comparison with " + single\_val + " metric",\

fontsize=**25**, color="blue") # setting the plot title.

plt.plot(k\_list,uniform , "--", marker = "o", label="weights uniform", color="red") # setting the uniform

# variables values to the plot on the y axis and K\_list on the x axis.

plt.plot(k\_list,distance ,":", marker = "\*", label="weights distant", color="blue") # setting the distant

# variables values to the plot on the y axis and K\_list on the x axis.

plt.xlabel("Number of nearest neighbours",fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='lower left') # setting the legend.

plt.grid() # setting a grid

# Sub function to preform Grid Search

**def** **Grid\_search\_fun**(typeofmodelandprams, dict\_prams, crossval, X, y): # Function takes in the model type,

# number of crossvalidation sand X and y values as hyperparameters.

"""

The function Grid\_search\_fun performs grid search on a model. It prints the best results and

the hyperparametrs used to obtain those results and then it plots those results.

The first parameter typeofmodelandprams is the type of model used on which random search is performed on.

The secound parameter dict\_prams is a dictinory of hyperparameters to be used in the random search.

The third parameter crossval is the number of cross validations to be performed.

The fourth parameter X is the features.

The fifth parameter y is the labels.

The function Grid\_search\_fun returns the results of the gird search.

"""

model = typeofmodelandprams # creating an instance of the object.

parameters = [dict\_prams] # hyper parameters for the grid search

grid\_search = GridSearchCV(estimator = model, # model

param\_grid = parameters, # hyper paramaters

scoring = 'accuracy', # score measurement

cv = crossval, # number of cross validations

n\_jobs = -**1**, return\_train\_score=False) # selecting all possible paramaters to go

# through to get the best model possible # train score is false as it can be computationaly expensive.

# without storing the traning score the grd search is fater

grid\_search.fit(X, y) # applying the search on our model.

#print(pd.DataFrame(grid\_search.cv\_results\_)[["mean\_test\_score","params"]])

**print**(pd.DataFrame(grid\_search.cv\_results\_)) # to print the whole result

best\_accuracy = grid\_search.best\_score\_ # the best accuracy

best\_parameters = grid\_search.best\_params\_ # the best paramaters that gave the best accurecy

**print**("Best Accuracy: {:.2f} %".format(best\_accuracy\***100**)) # printing best accuracy

**print**("Best Parameters:", best\_parameters) # printing the best parameters

**return** grid\_search # returns grid search value

# Performs full grid search for KNN model

**def** **KNN\_full\_grid\_search**(X, y): # Performs grid search.

"""

The function KNN\_full\_grid\_search performs grid search on KNN model. It prints the best results and

the hyperparametrs used to obtain those results and then it plots those results.

The first parameter X is the features.

The secound parameter y is the labels.

"""

metric\_list = ['minkowski','euclidean']

**for** single\_val **in** metric\_list:

k\_list = list(range(**1**,**31**)) # list from 1 - 31 (will be used as KNN's)

weight\_list = ["uniform", "distance"] # weights list uniform and distance

para\_grid = dict(n\_neighbors=k\_list,weights=weight\_list,metric=[single\_val])

# adding the above lists in a dictinary.

grid\_search = Grid\_search\_fun(KNeighborsClassifier(n\_neighbors=k\_list), para\_grid, **5**, X, y) # Using the grid

# search function with arguments given

grid\_search.cv\_results\_["mean\_test\_score"] # looking at mean test score

uniform = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all even number

# indexes using a for loop.

**if** grid\_search.cv\_results\_["param\_weights"][i] == "uniform":

uniform.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the unifrom values to the

# varibale unifrom.

distance = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search.cv\_results\_["param\_weights"][i] == "distance":

distance.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the distance values to the

# varibale distance.

grid\_search\_plot(distance, uniform, k\_list, single\_val)

# X and y un\_edited

## Grid Search

KNN\_full\_grid\_search(X, y) # performs grid search.

# Over Sampled X and y values

## Grid Search

KNN\_full\_grid\_search(X\_over, y\_over) # performs grid search.

# X and y obtained by using Corelation matrix

## Grid Search

KNN\_full\_grid\_search(X2, y) # performs grid search.

# Model Evaluation using hyperparameters obtained using.

# Cross Valadation

# Random Search

# Grid Search

# X and y un\_edited

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

**for** neighbour **in** [**1**, **11**, **29**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

Classifier\_function(KNeighborsClassifier(weights = 'uniform', n\_neighbors = neighbour), X\_train,\

y\_train,X\_test, y\_test, "KNN Model")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Results after feature scaling

# StandardScaler

**for** neighbour **in** [**1**, **11**, **29**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(StandardScaler(), X\_train, X\_test, y\_train, y\_test,\

KNeighborsClassifier(weights = 'uniform', n\_neighbors = neighbour) ,"KNN Model")

# performs standard scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# MinMaxScaler

**for** neighbour **in** [**1**, **11**, **29**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(MinMaxScaler(), X\_train, X\_test, y\_train, y\_test,\

KNeighborsClassifier(weights = 'uniform', n\_neighbors = neighbour) ,"KNN Model")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# RobustScaler

**for** neighbour **in** [**1**, **11**, **29**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(RobustScaler(), X\_train, X\_test, y\_train, y\_test,\

KNeighborsClassifier(weights = 'uniform', n\_neighbors = neighbour) ,"KNN Model")

# performs robust scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Normalizer

**for** neighbour **in** [**1**, **11**, **29**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(Normalizer(), X\_train, X\_test, y\_train, y\_test,\

KNeighborsClassifier(weights = 'uniform', n\_neighbors = neighbour) ,"KNN Model")

# Performs normalization scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Over Sampled X and y values

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

**for** neighbour **in** [**1**, **3**, **15**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

Classifier\_function(KNeighborsClassifier(n\_neighbors = neighbour, weights ='distance'), X\_train\_over,\

y\_train\_over, X\_test\_over, y\_test\_over, "KNN Model")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Feature scaling

# StandardScaler

**for** neighbour **in** [**1**, **3**, **15**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(StandardScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

KNeighborsClassifier(n\_neighbors = neighbour, weights ='distance') ,"KNN Model")

# performs standard scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# MinMaxScaler

**for** neighbour **in** [**1**, **3**, **15**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(MinMaxScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

KNeighborsClassifier(n\_neighbors = neighbour, weights ='distance') ,"KNN Model")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# RobustScaler

**for** neighbour **in** [**1**, **3**, **15**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(RobustScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

KNeighborsClassifier(n\_neighbors = neighbour, weights ='distance') ,"KNN Model")

# performs robust scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Normalizer

**for** neighbour **in** [**1**, **3**, **15**]: # looping from a list.

**print**("When neighbour = ", neighbour) # printing output

featurescaling(Normalizer(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

KNeighborsClassifier(n\_neighbors = neighbour, weights ='distance') ,"KNN Model")

# Performs normalization scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# X and y obtained by using Corelation matrix

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

**for** neighbour **in** [**17**,**19**]: # looping through a list

**print**("When neighbour = ", neighbour) # printing output

Classifier\_function(KNeighborsClassifier(n\_neighbors = neighbour, weights = "uniform"), X2\_train, y2\_train,\

X2\_test, y2\_test, "KNN Model")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

leaf\_size\_list = [**24**,**25**,**26**,**27**,**28**,**29**] # creating a list

**for** lsz **in** leaf\_size\_list: # looping through a list

**print**("When Leaf Size = ", lsz) # printing output

crossvalscore(KNeighborsClassifier(weights = 'uniform', n\_neighbors = **19**,algorithm = 'ball\_tree', \

leaf\_size = lsz), X, y, **10**) # Performs cross validation.

**print**("**\n**") # printing new line

leaf\_size\_list = [**24**,**25**,**26**,**27**,**28**,**29**] # creating a list

**for** lsz **in** leaf\_size\_list: # looping through a list

**print**("When Leaf Size = ", lsz) # printing output

Classifier\_function(KNeighborsClassifier(n\_neighbors = **19**, weights = "uniform", algorithm = 'ball\_tree', \

leaf\_size = lsz), X2\_train, y2\_train,X2\_test, y2\_test, "KNN Model")

**print**("**\n**") # printing new line

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Results

# Original results without hyperparameter Improvement

Classifier\_function(KNeighborsClassifier(), X\_train, y\_train,X\_test, y\_test, "KNN Model")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Best Score and model

# Over Sampled X and y values

# MinMaxScaler

# results with hyparameters improved

featurescaling(MinMaxScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

KNeighborsClassifier(n\_neighbors = **3**, weights ='distance') ,"KNN Model")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

featurescaling(MinMaxScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

KNeighborsClassifier(n\_neighbors = **15**, weights ='distance') ,"KNN Model")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Checking Best Results with different Random\_states

sucidedataframe.info() # checking Basic information on the dataframe being procesed.

# Un edited full evalution

crossvalscore(KNeighborsClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

# Improved model full Evaluation with feature scaling

# With 15 neighbours

crossvalscore(KNeighborsClassifier(n\_neighbors = **15**, weights ='distance'), X\_over, y\_over, **10**) # Performs cross validation.

# With 3 neighbours

crossvalscore(KNeighborsClassifier(n\_neighbors = **3**, weights ='distance'), X\_over, y\_over, **10**) # Performs cross validation.

# With 1 neighbour

crossvalscore(KNeighborsClassifier(n\_neighbors = **1**, weights ='distance'), X\_over, y\_over, **10**) # Performs cross validation.

# Random forest Model

## k-fold cross-validation Graph

**def** **subplot**(list\_for\_plot, ave\_scores, hyper\_pram\_to\_use\_name): # A function with parameters given.

"""

The function subplot plots a linear plot and shows the best results using two lists

the fist one acting as a label to the second list and the second list being scores.

The first parameter list\_for\_plot is the X label.

The second parameter ave\_scores is the y label.

The thrid paramater hyper\_pram\_to\_use\_name is the the titile for the plot.

"""

**print**(hyper\_pram\_to\_use\_name + " ", list\_for\_plot,"**\n**Average " + hyper\_pram\_to\_use\_name +\

" scores", ave\_scores) # printing the average score of the model

**for** i **in** range(len(ave\_scores)): # looping to the legth of list

**if** ave\_scores[i] == max(ave\_scores): # if the first value is equal to the second value

**print**("The hyperparameter",list\_for\_plot[i], "gives the best result :", ave\_scores[i]) # print output.

**print**("**\n\n**") # print new line

plt.figure() # plot a figure

plt.title("Best " + hyper\_pram\_to\_use\_name + " Selection",fontsize=**18**) # Displays plot title

plt.plot(list\_for\_plot, ave\_scores) # Displays description of the plots x and y labels.

plt.xlabel(hyper\_pram\_to\_use\_name +" values") # Displays the x axis for the plot

plt.ylabel("Average CV model accuracy") # Displays the y axis for the plot

plt.legend([hyper\_pram\_to\_use\_name], loc="lower right") # adds a legend to the plot.

plt.grid() # adds a gird to the plot

**def** **RandomForest\_plot\_Values**(X,y, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list):

# Values required for plotting

"""

The RandomForest\_plot\_Values performs cross valadation on the Random Forest models provided hyperparametes, prints

the best hyperparameters that give the best reults and plots the results.

The first parameter X takes the features.

The first parameter y takes the lables.

The third parameter n\_estimators\_list is a list of n\_estimators.

The fourth parameter max\_depth\_list is a list of max\_depth.

The fifth parameter min\_samples\_leaf\_list is a list of min\_samples\_leaf.

The sixth parameter min\_samples\_split\_list is a list of min\_samples\_split.

"""

ave\_scores = [] # Creating a empty list

**for** single\_val **in** n\_estimators\_list: # looping through each value in the list

model = RandomForestClassifier(n\_estimators=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(n\_estimators\_list, ave\_scores, "n\_estimators") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** max\_depth\_list: # looping through each value in the list

model = RandomForestClassifier(max\_depth=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(max\_depth\_list, ave\_scores, "max\_depth") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** min\_samples\_leaf\_list: # looping through each value in the list

model = RandomForestClassifier(min\_samples\_leaf=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(min\_samples\_leaf\_list, ave\_scores, "min\_samples\_leaf") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** min\_samples\_split\_list: # looping through each value in the list

model = RandomForestClassifier(min\_samples\_split=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(min\_samples\_split\_list, ave\_scores, "min\_samples\_split") # using the function to plot x and y values.

## Plotting the Random Forest with a average Cross-val score

n\_estimators\_list = [**10**,**50**,**100**,**200**, **300**, **500**] # Setting the values for the list.

max\_depth\_list = [None, **4**,**5**,**6**,**7**,**8** ,**80**, **90**, **100**, **200**,**500**] # Setting the values for the list.

min\_samples\_leaf\_list = list(range(**1**,**30**)) # Setting the values for the list.

min\_samples\_split\_list = list(range(**2**,**30**)) # Setting the values for the list.

# X and y un\_edited

RandomForest\_plot\_Values(X,y, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list)

# PLots KNN Cross Val score

# Over Sampled X and y values

## Plotting the KNN's with a average Cross-val score

RandomForest\_plot\_Values(X\_over,y\_over, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list)

# PLots Cross Val score

# X and y obtained by using Corelation matrix

## Plotting the Random Forest with a average Cross-val score

RandomForest\_plot\_Values(X2,y, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list)

# PLots Cross Val score

**def** **n\_estimation**(X,y): # a function with parameters given

"""

The n\_estimation performs cross valadation on the Random Forest models n\_estimators parameter prints

the best hyperparameters that give the best reults and plots the results

The first parameter X takes the features.

The first parameter y takes the lables.

"""

n\_estimators\_list = [**10**,**15**,**25**,**30**,**45**,**50**,**60**,**75**,**90**,**95**,**100**,**105**,**110**,**125**,**130**,**145**,**150**,**160**,**175**,**190**,**200**,**210**,\

**225**,**230**,**250**,**275**,**300**] # a list of n\_estimators.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** n\_estimators\_list: # looping through each value in the list

model = RandomForestClassifier(n\_estimators=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(n\_estimators\_list, ave\_scores, "n\_estimators") # using the function to plot x and y values.

# X and y un\_edited

# N estimation improvement

n\_estimation(X,y) # PLots Cross Val score

# Over Sampled X and y values

# N estimation improvement

n\_estimation(X\_over,y\_over) # PLots Cross Val score

# X and y obtained by using Corelation matrix

# N estimation improvement

n\_estimation(X2,y) # PLots Cross Val score

**def** **max\_depth\_estimation**(X,y):

"""

The max\_depth\_estimation( performs cross valadation on the Random Forest models max\_depth parameter prints

the best hyperparameters that give the best reults and plots the results

The first parameter X takes the features.

The first parameter y takes the lables.

"""

max\_depth\_list = [None, **1**, **5**,**10**, **15**,**20** ,**25**, **30**,**40**,**45**,**50**, **60**,**70**,**75**, **80**, **85**,**90**,**95**,**100**] # a list of max\_depth values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** max\_depth\_list: # looping through each value in the list

model = RandomForestClassifier(max\_depth=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(max\_depth\_list, ave\_scores, "max\_depth") # using the function to plot x and y values.

# X and y un\_edited

# Max depth improvement

max\_depth\_estimation(X,y)

# PLots Cross Val score

# Over Sampled X and y values

# Max depth improvement

max\_depth\_estimation(X\_over,y\_over) # PLots Cross Val score

# X and y obtained by using Corelation matrix

# Max depth improvement

max\_depth\_estimation(X2,y) # PLots Cross Val score

# n\_estimators = [10, 75, 150, 225]

# max\_depth = [10, 50 ,100, 75 , 90]

# min\_samples\_leaf = [1,9, 10]

# min\_samples\_split = [2,16, 20]

# X and y un\_edited

# Without any model Tuning

crossvalscore(RandomForestClassifier(), X, y, **10**) # Performs cross validation.

# Using Cross Validation hyperparameter selection

n\_list = [**110**, **190**, **300**] # a list of n\_estimators

**for** i **in** n\_list: # looping through a list

**print**("Result when n\_estimators = ", i) # printing output

crossvalscore(RandomForestClassifier(n\_estimators = i, max\_depth = **10**, min\_samples\_leaf = **18**, min\_samples\_split =**19**\

), X, y, **10**) # Performs cross validation.

**print**("**\n**") # priting a new line.

# Over Sampled X and y values

# Without any model Tuning

crossvalscore(RandomForestClassifier(), X\_over, y\_over, **10**)

# Performs cross validation.

# Using Cross Validation hyperparameter selection

n\_list = [**90**, **95**, **100**] # a list of n\_estimators

md\_list = [None, **50**, **75**] # a list of max\_depth

**for** n **in** n\_list: # looping through a list

**for** md **in** md\_list: # looping through a list

**print**("Result when n\_estimators = ", n) # printing output

**print**("Result when max depth = ", md) # printing output

crossvalscore(RandomForestClassifier(n\_estimators = n, max\_depth = md, min\_samples\_leaf = **1**, min\_samples\_split =**2**\

), X\_over, y\_over, **10**) # Performs cross validation.

**print**("**\n**") # printing new line

# X and y obtained by using Corelation matrix

# Without any model Tuning

crossvalscore(RandomForestClassifier(), X2, y, **10**)

# Performs cross validation.

# Using Cross Validation hyperparameter selection

n\_list = [**110**, **230**, **250**] # a list of n\_estimators

mss\_list = [**11**, **18**] # a list of n\_estimators

**for** i **in** n\_list: # looping through a list

**for** j **in** mss\_list: # looping through a list

**print**("Result when n\_estimators = ", i) # printing output

**print**("Result when min\_sample\_split = ", j) # printing output

crossvalscore(RandomForestClassifier(n\_estimators =i, max\_depth = **25**, min\_samples\_leaf = **12**, \

min\_samples\_split =j ), X2, y, **10**)

# Performs cross validation.

**print**("**\n**") # printing new line

# Full random search on random forest

**def** **randomsearchrandomforest**(X,y,n\_estimators\_list,max\_depth\_list,min\_samples\_leaf\_list,min\_samples\_split\_list):

"""

The function randomsearchrandomforest performs random search on a model. It prints the best results and

the hyperparametrs used to obtain those results and then it plots those results.

The first parameter X is the feature.

The secound parameter y is the labels

The third parameter n\_estimators\_list is the n\_estimators.

The fourth parameter max\_depth\_list is the max\_depth.

The fifth parameter min\_samples\_leaf\_list is the min\_samples\_leaf.

The sixth parameter min\_samples\_split\_list is the min\_samples\_split.

"""

bootstrap\_list = [True, False] # a lsit with values specified.

max\_features\_list = ['auto', 'sqrt', 'log2'] # Setting the values for the list.

criterion\_list = ['gini', 'entropy'] # Setting the values for the list.

**for** i **in** range(len(criterion\_list)):

**for** j **in** range(len(max\_features\_list)):

list1 = [] # creating a empty list

list2 = [] # creating a empty list

para\_rand = dict(n\_estimators = n\_estimators\_list, max\_depth = max\_depth\_list, min\_samples\_leaf = \

min\_samples\_leaf\_list,min\_samples\_split = min\_samples\_split\_list,\

max\_features = [max\_features\_list[j]], criterion = [criterion\_list[i]],\

bootstrap = bootstrap\_list) # adding the above

# lists in a dictinary.

rand\_search = rand\_search\_fun(RandomForestClassifier(), para\_rand, **5**, X, y) # Using the grid search gunction with

## Random search # Using the grid search gunction with

**for** m **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times the length of

# the mean\_test\_score

list1.append(rand\_search.cv\_results\_["mean\_test\_score"][m]) # appending the values to list

list2.append(rand\_search.cv\_results\_["param\_bootstrap"][m]) # appedning the values to the list2.

sorted\_list1= []

sorted\_list2 = list2.copy() # creating a copy of the list2 variables values.

sorted\_list2.sort() # sorting the list2

**for** k **in** range(len(sorted\_list2)): # Looping through the lenght of the sorted\_list2 variable.

**for** l **in** range(len(list2)): # Looping through the lenght of the list2 variable.

**if** sorted\_list2[k] == list2[l]: # checking that if the values of sorted\_list2[i]

# is equal to list2[j]

sorted\_list1.append(list1[l]) # appeding the list1[j] values to the sorted\_list1 list.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Random Search Hyper parameter Comparison with "+ criterion\_list[i] +" criterion", \

fontsize=**25**, color="blue") # setting the plot title.

plt.plot(sorted\_list2,sorted\_list1 , "--", marker = "o", label="max features " + max\_features\_list[j],\

color="red") # setting the best

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('bootstrap',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# X and y un\_edited

## Random search

n\_estimators\_list = [**110**] # a lsit with values specified.

max\_depth\_list = [**10**] # a lsit with values specified.

min\_samples\_leaf\_list = [**18**] # a lsit with values specified.

min\_samples\_split\_list = [**19**] # a lsit with values specified.

randomsearchrandomforest(X,y,n\_estimators\_list,max\_depth\_list,min\_samples\_leaf\_list,min\_samples\_split\_list) # function with arguments given

# Over Sampled X and y values

## Random search

n\_estimators\_list = [**100**] # a lsit with values specified.

max\_depth\_list = [**50**] # a lsit with values specified.

min\_samples\_leaf\_list = [**1**] # a lsit with values specified.

min\_samples\_split\_list = [**2**] # a lsit with values specified.

randomsearchrandomforest(X\_over,y\_over,n\_estimators\_list,max\_depth\_list,min\_samples\_leaf\_list,min\_samples\_split\_list) # function with arguments given

# X and y obtained by using Corelation matrix

## Random search

n\_estimators\_list = [**230**] # a lsit with values specified.

max\_depth\_list = [**18**] # a lsit with values specified.

min\_samples\_leaf\_list = [**12**] # a lsit with values specified.

min\_samples\_split\_list = [**18**] # a lsit with values specified.

randomsearchrandomforest(X2,y,n\_estimators\_list,max\_depth\_list,min\_samples\_leaf\_list,min\_samples\_split\_list) # function with arguments given

# Full Grid search on random forest

**def** **randomforestgridsearch**(X, y, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list): # function

# with parameters given

"""

The function randomforestgridsearch performs grid search on a model. It prints the best results and

the hyperparametrs used to obtain those results and then it plots those results.

The first parameter X is the feature.

The secound parameter y is the labels

The third parameter n\_estimators\_list is the n\_estimators.

The fourth parameter max\_depth\_list is the max\_depth.

The fifth parameter min\_samples\_leaf\_list is the min\_samples\_leaf.

The sixth parameter min\_samples\_split\_list is the min\_samples\_split.

"""

bootstrap\_list = [True, False] # a lsit with values specified.

max\_features\_list = ['auto', 'sqrt', 'log2'] # Setting the values for the list.

criterion\_list = ['gini', 'entropy'] # Setting the values for the list.

**for** i **in** range(len(criterion\_list)):

**if** i == **0**:

para\_grid = dict(n\_estimators = n\_estimators\_list, max\_depth = max\_depth\_list, min\_samples\_leaf = \

min\_samples\_leaf\_list,\

min\_samples\_split = min\_samples\_split\_list,max\_features = max\_features\_list, \

criterion = [criterion\_list[i]], bootstrap = bootstrap\_list) # adding the above

# lists in a dictinary.

grid\_search = Grid\_search\_fun(RandomForestClassifier(), para\_grid, **5**, X, y) # Using the grid search function with

**if** i == **1**:

para\_grid = dict(n\_estimators = n\_estimators\_list, max\_depth = max\_depth\_list, min\_samples\_leaf = \

min\_samples\_leaf\_list,min\_samples\_split = min\_samples\_split\_list,\

max\_features = max\_features\_list,criterion = [criterion\_list[i]],\

bootstrap = bootstrap\_list) # adding the above

# lists in a dictinary.

grid\_search2 = Grid\_search\_fun(RandomForestClassifier(), para\_grid, **5**, X, y) # Using the grid search function with

list1 = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all even number

# indexes using a for loop.

**if** grid\_search.cv\_results\_["param\_max\_features"][i] == 'auto':

list1.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the 32 values to the

# varibale list1.

list2 = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search.cv\_results\_["param\_max\_features"][i] == 'sqrt':

list2.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the 33 values to the

# varibale list2.

list3 = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search.cv\_results\_["param\_max\_features"][i] == 'log2':

list3.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the 34 values to the

# varibale lst3.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Grid Search Hyper parameter Comparison with gini criterion", fontsize=**25**, color="blue")

# setting the plot title.

plt.plot(bootstrap\_list,list1 , "--", marker = "o", label="max features list auto", color="red") # setting the best

plt.plot(bootstrap\_list,list2 ,":", marker = "\*", label="max features list sqrt", color="blue") # setting the random

plt.plot(bootstrap\_list,list3 ,":", marker = "d", label="max features list log2", color="orange") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('bootstrap',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

list1 = [] # a empty list is created

**for** i **in** range(len(grid\_search2.cv\_results\_["mean\_test\_score"])): # getting all even number

# indexes using a for loop.

**if** grid\_search2.cv\_results\_["param\_max\_features"][i] == 'auto':

list1.append(grid\_search2.cv\_results\_["mean\_test\_score"][i]) # appending the 32 values to the

# varibale list1.

list2 = [] # a empty list is created

**for** i **in** range(len(grid\_search2.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search2.cv\_results\_["param\_max\_features"][i] == 'sqrt':

list2.append(grid\_search2.cv\_results\_["mean\_test\_score"][i]) # appending the 33 values to the

# varibale list2.

list3 = [] # a empty list is created

**for** i **in** range(len(grid\_search2.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search2.cv\_results\_["param\_max\_features"][i] == 'log2':

list3.append(grid\_search2.cv\_results\_["mean\_test\_score"][i]) # appending the 34 values to the

# varibale list3.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Grid Search Hyper parameter Comparison with 31 min samples split", fontsize=**25**, color="blue")

# setting the plot title.

plt.plot(bootstrap\_list,list1 , "--", marker = "o", label="max features list auto", color="red") # setting the best

plt.plot(bootstrap\_list,list2 ,":", marker = "\*", label="max features list sqrt", color="blue") # setting the random

plt.plot(bootstrap\_list,list3 ,":", marker = "d", label="max features list log2", color="orange") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('bootstrap',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# X and y un\_edited

## Grid Search

n\_estimators\_list = [**110**] # a lsit with values specified.

max\_depth\_list = [**10**] # a lsit with values specified.

min\_samples\_leaf\_list = [**18**] # a lsit with values specified.

min\_samples\_split\_list = [**19**] # a lsit with values specified.

randomforestgridsearch(X, y, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list) # function

# with arguments given.

# Over Sampled X and y values

## Grid Search

n\_estimators\_list = [**90**] # a lsit with values specified.

max\_depth\_list = [**50**] # a lsit with values specified.

min\_samples\_leaf\_list = [**1**] # a lsit with values specified.

min\_samples\_split\_list = [**2**] # a lsit with values specified.

randomforestgridsearch(X\_over, y\_over, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list) # function

# with arguments given.

# X and y obtained by using Corelation matrix

## Grid Search

n\_estimators\_list = [**230**] # a lsit with values specified.

max\_depth\_list = [**18**] # a lsit with values specified.

min\_samples\_leaf\_list = [**12**] # a lsit with values specified.

min\_samples\_split\_list = [**18**] # a lsit with values specified.

randomforestgridsearch(X, y, n\_estimators\_list, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list) # function

# with arguments given.

# X and y un\_edited

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

Classifier\_function(RandomForestClassifier(n\_estimators =**110**,min\_samples\_split = **19**,\

min\_samples\_leaf = **18**, max\_depth = **10**, bootstrap = True, \

max\_features = 'sqrt', criterion = 'entropy' ), X\_train, y\_train,X\_test, y\_test, "Random Forest")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Results after feature scaling

# StandardScaler

featurescaling(StandardScaler(), X\_train, X\_test, y\_train, y\_test, \

RandomForestClassifier(n\_estimators =**110**,min\_samples\_split = **19**,min\_samples\_leaf = **18**, \

max\_depth = **10**, bootstrap = True, max\_features = 'sqrt', criterion = 'entropy' ),\

"Random Forest")

# performs standard scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# MinMaxScaler

featurescaling(MinMaxScaler(), X\_train, X\_test, y\_train, y\_test, \

RandomForestClassifier(n\_estimators =**110**,min\_samples\_split = **19**,min\_samples\_leaf = **18**, \

max\_depth = **10**, bootstrap = True, max\_features = 'sqrt', criterion = 'entropy' ),\

"Random Forest")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# RobustScaler

featurescaling(RobustScaler(), X\_train, X\_test, y\_train, y\_test, \

RandomForestClassifier(n\_estimators =**110**,min\_samples\_split = **19**,min\_samples\_leaf = **18**, \

max\_depth = **10**, bootstrap = True, max\_features = 'sqrt', criterion = 'entropy' ),\

"Random Forest")

# performs robust scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Normalizer

featurescaling(Normalizer(), X\_train, X\_test, y\_train, y\_test, \

RandomForestClassifier(n\_estimators =**110**,min\_samples\_split = **19**,min\_samples\_leaf = **18**, \

max\_depth = **10**, bootstrap = True, max\_features = 'sqrt', criterion = 'entropy' ),\

"Random Forest")

# Performs normalization scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Over Sampled X and y values

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

Classifier\_function(RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**,\

min\_samples\_leaf = **1**, max\_depth = **50**, bootstrap = False, \

max\_features = 'sqrt', criterion = 'entropy'), X\_train\_over, y\_train\_over,X\_test\_over, \

y\_test\_over, "Random Forest")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Feature scaling

# StandardScaler

featurescaling(StandardScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**, min\_samples\_leaf = **1**, max\_depth = **50**,\

bootstrap = False, max\_features = 'sqrt', criterion = 'entropy') ,"Random Forest")

# performs standard scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# MinMaxScaler

featurescaling(MinMaxScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**, min\_samples\_leaf = **1**, max\_depth = **50**,\

bootstrap = False, max\_features = 'sqrt', criterion = 'entropy') ,"Random Forest")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# RobustScaler

featurescaling(RobustScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**, min\_samples\_leaf = **1**, max\_depth = **50**,\

bootstrap = False, max\_features = 'sqrt', criterion = 'entropy') ,"Random Forest")

# performs robust scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Normalizer

featurescaling(Normalizer(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**, min\_samples\_leaf = **1**, max\_depth = **50**,\

bootstrap = False, max\_features = 'sqrt', criterion = 'entropy') ,"Random Forest")

# Performs normalization scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# X and y obtained by using Corelation matrix

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

Classifier\_function(RandomForestClassifier(n\_estimators =**230**,min\_samples\_split = **2**,\

min\_samples\_leaf = **12**, max\_depth = **18**, bootstrap = False, \

max\_features = 'sqrt', criterion = 'entropy'), X2\_train, y2\_train,X2\_test, y2\_test, "Random Forest")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Results

# Original results without hyperparameter Improvement

Classifier\_function(RandomForestClassifier(), X\_train, y\_train,X\_test, y\_test, "Random Forest")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Best Score and model

# Over Sampled X and y values

# MinMaxScaler

# Results with hyparameters improved

featurescaling(MinMaxScaler(), X\_train\_over, X\_test\_over, y\_train\_over, y\_test\_over,\

RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**, min\_samples\_leaf = **1**, max\_depth = **50**,\

bootstrap = False, max\_features = 'sqrt', criterion = 'entropy') ,"Random Forest")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Un\_edited full model evaluation

crossvalscore(RandomForestClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

# Improved model full Evaluation with feature scaling

crossvalscore(RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**, min\_samples\_leaf = **1**,\

max\_depth = **50**,bootstrap = False, max\_features = 'sqrt', criterion = 'entropy'), X\_over, y\_over, **10**)

# Performs cross validation.

# Decision Tree

**def** **DecisionTree\_plot\_Values**(X,y, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list, min\_weight\_fraction\_leaf\_list,\

max\_leaf\_nodes\_list, criterion\_list,max\_features\_list, splitter\_list):

# Values required for plotting

"""

The DecisionTree\_plot\_Values performs cross valadation on the Decision Tree models provided hyperparametes, prints

the best hyperparameters that give the best reults and plots the results.

The first parameter X takes the features.

The first parameter y takes the lables.

The third parameter max\_depth\_list is a list of max\_depth.

The fourth parameter min\_samples\_leaf\_list is a list of min\_samples\_leaf.

The fifth parameter min\_weight\_fraction\_leaf\_list is a min\_weight\_fraction\_leaf.

The sixth parameter max\_leaf\_nodes\_list is a list of max\_leaf\_nodes.

The seventh parameter criterion\_list is a list of criterion.

The eigth parameter max\_features\_list is a max\_features.

The ningth parameter splitter\_list is a list of splitter.

"""

ave\_scores = [] # Creating a empty list

**for** single\_val **in** max\_depth\_list: # looping through each value in the list

model = DecisionTreeClassifier(max\_depth=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(max\_depth\_list, ave\_scores, "max\_depth") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** min\_samples\_leaf\_list: # looping through each value in the list

model = DecisionTreeClassifier(min\_samples\_leaf=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(min\_samples\_leaf\_list, ave\_scores, "min\_samples\_leaf") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** min\_samples\_split\_list: # looping through each value in the list

model = DecisionTreeClassifier(min\_samples\_split=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(min\_samples\_split\_list, ave\_scores, "min\_samples\_split") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** min\_weight\_fraction\_leaf\_list: # looping through each value in the list

model = DecisionTreeClassifier(min\_weight\_fraction\_leaf=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(min\_weight\_fraction\_leaf\_list, ave\_scores, "min\_weight\_fraction\_leaf") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** max\_leaf\_nodes\_list: # looping through each value in the list

model = DecisionTreeClassifier(max\_leaf\_nodes=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(max\_leaf\_nodes\_list, ave\_scores, "max\_leaf\_nodes") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** criterion\_list: # looping through each value in the list

model = DecisionTreeClassifier(criterion=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(criterion\_list, ave\_scores, "criterion") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** max\_features\_list: # looping through each value in the list

model = DecisionTreeClassifier(max\_features=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

max\_features\_list[**3**] = "None"

subplot(max\_features\_list, ave\_scores, "max\_features") # using the function to plot x and y values.

ave\_scores = [] # Creating a empty list

**for** single\_val **in** splitter\_list: # looping through each value in the list

model = DecisionTreeClassifier(splitter=single\_val) # creating an instance of a class.

scores = cross\_val\_score(model,X,y,cv=**5**) # Getting the results of the model.

ave\_scores.append(round(scores.mean(),**3**)) # getting the average score from the model and appending

# it to the ave\_scores list.

subplot(splitter\_list, ave\_scores, "splitter") # using the function to plot x and y values.

# X and y un\_edited

## Plotting the KNN's with a average Cross-val score

max\_depth\_list = [None,**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**,**10**,**25**,**50**,**75**,**80**,**90**,**100**,**125**,**150**,**175**,**200**,**225**,**250**,**275**,**300**,**500**] # a lsit with values

# specified.

min\_samples\_leaf\_list = list(range(**1**,**50**)) # a lsit with values specified.

min\_samples\_split\_list = list(range(**2**,**50**)) # a lsit with values specified.

min\_weight\_fraction\_leaf\_list = [**0.0**, **0.1**,**0.2**,**0.3**,**0.4**,**0.5**,**0.6**] # a lsit with values specified.

max\_leaf\_nodes\_list = [None,**10**,**20**,**30**,**40**,**50**,**60**,**70**,**80**,**90**,**100**,**125**, **150**,**175**, **200**,**225**,**250**,**275**,**300**] # a lsit with values specified.

min\_impurity\_decrease = [**0.0**] # a lsit with values specified.

criterion\_list = ["gini", "entropy"] # a lsit with values specified.

max\_features\_list = ["auto","log2","sqrt", None] # a lsit with values specified.

splitter\_list = ["best","random"] # a lsit with values specified.

DecisionTree\_plot\_Values(X,y, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list, min\_weight\_fraction\_leaf\_list, \

max\_leaf\_nodes\_list, criterion\_list,max\_features\_list, splitter\_list) # PLots Cross Val score

# Over Sampled X and y values

## Plotting the KNN's with a average Cross-val score

DecisionTree\_plot\_Values(X\_over,y\_over, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list, \

min\_weight\_fraction\_leaf\_list, max\_leaf\_nodes\_list, criterion\_list,max\_features\_list, splitter\_list) # PLots Cross

# Val score

# X and y obtained by using Corelation matrix

## Plotting the KNN's with a average Cross-val score

DecisionTree\_plot\_Values(X2,y, max\_depth\_list, min\_samples\_leaf\_list, min\_samples\_split\_list, min\_weight\_fraction\_leaf\_list, \

max\_leaf\_nodes\_list, criterion\_list,max\_features\_list, splitter\_list)

# PLots Cross Val score

# X and y un\_edited

# Without any model Tuning

crossvalscore(DecisionTreeClassifier(), X, y, **10**) # Performs cross validation.

# Using Cross Validation hyperparameter selection

msl\_list = [**32**, **33**, **34**] # a list of min\_samples\_leaf

mss\_list = [**31**, **29**] # a list of min\_samples\_split

**for** msl **in** msl\_list: # looping through a list

**for** mss **in** mss\_list: # looping through a list

**print**("Result when min\_samples\_leaf = ", msl) # printing output

**print**("Result when min\_samples\_split = ", mss) # printing output

crossvalscore(DecisionTreeClassifier(min\_samples\_leaf = msl, min\_samples\_split = mss, max\_leaf\_nodes = **10**, \

min\_weight\_fraction\_leaf = **0.0**, max\_depth = **4**, criterion = 'entropy', \

max\_features = "auto", splitter = "random"), X, y, **10**)

# gives Cross Val score

**print**("**\n**") # printing a new line

# Over Sampled X and y values

# Without any model Tuning

crossvalscore(DecisionTreeClassifier(), X\_over, y\_over, **10**)

# Performs cross validation.

# Using Cross Validation hyperparameter selection

crossvalscore(DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**, max\_leaf\_nodes = None, \

min\_weight\_fraction\_leaf = **0.0**, max\_depth = **125**, criterion = 'entropy', \

max\_features = None, splitter = "random"), X\_over, y\_over, **10**)

# Performs cross validation.

# X and y obtained by using Corelation matrix

# Without any model Tuning

crossvalscore(DecisionTreeClassifier(), X2, y, **10**) # Performs cross validation.

# Using Cross Validation hyperparameter selection

crossvalscore(DecisionTreeClassifier(min\_samples\_leaf = **10**, min\_samples\_split = **30**, max\_leaf\_nodes = **20**, \

min\_weight\_fraction\_leaf = **0.0**, max\_depth = **3**, criterion = 'entropy', \

max\_features = "sqrt", splitter = "random"), X2, y, **10**)

# Performs cross validation.

# X and y un\_edited

## Random search

max\_leaf\_nodes\_list = list(range(**1**,**20**)) # a lsit with values specified.

min\_samples\_leaf\_list = [**32**, **33**, **34**] # a lsit with values specified.

min\_samples\_split\_list = [**29**, **31**] # a lsit with values specified.

**for** i **in** range(len(min\_samples\_split\_list)):

**for** j **in** range(len(min\_samples\_leaf\_list)):

list1 = [] # creating a empty list

list2 = [] # creating a empty list

para\_rand = dict(max\_leaf\_nodes = max\_leaf\_nodes\_list, min\_samples\_leaf = [min\_samples\_leaf\_list[j]], \

min\_samples\_split = [min\_samples\_split\_list[i]]) # adding the above

# lists in a dictinary.

rand\_search = rand\_search\_fun(DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**, \

max\_depth = **4**, criterion = 'entropy', \

max\_features = "auto", splitter = "random"), \

para\_rand, **5**, X, y) # X and y un\_edite

## Random search # Using the grid search gunction with

**for** m **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times

# the length of

# the mean\_test\_score

list1.append(rand\_search.cv\_results\_["mean\_test\_score"][m]) # appending the values to list

list2.append(rand\_search.cv\_results\_["param\_max\_leaf\_nodes"][m]) # appedning the

# values to the list2.

sorted\_list1= []

sorted\_list2 = list2.copy() # creating a copy of the list2 variables values.

sorted\_list2.sort() # sorting the list2

**for** k **in** range(len(sorted\_list2)): # Looping through the lenght of the sorted\_list2 variable.

**for** l **in** range(len(list2)): # Looping through the lenght of the list2 variable.

**if** sorted\_list2[k] == list2[l]: # checking that if the values of

# sorted\_list2[i] is equal to list2[j]

sorted\_list1.append(list1[l]) # appeding the list1[j] values to the sorted\_list1 list.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Random Search Hyper parameter Comparison with "+str(min\_samples\_split\_list[i]) +" **\**

min samples split", fontsize=**25**, color="blue") # setting the plot title.

plt.plot(sorted\_list2,sorted\_list1 , "--", marker = "o", label="min samples leaf " \

+ str(min\_samples\_leaf\_list[j]), color="red") # setting the best

# variables values to the plot on the y axis and

# max depth on the x axis.

plt.xlabel('max leaf nodes',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# Over Sampled X and y values

## Random search

max\_depth\_list = list(range(**101**,**150**)) # a lsit with values specified.

splitter\_list = ['best'] # a lsit with values specified.

para\_rand = dict(max\_depth = max\_depth\_list, splitter = splitter\_list) # adding the above

# lists in a dictinary.

rand\_search = rand\_search\_fun(DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**, max\_leaf\_nodes = None, \

min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', \

max\_features = None), para\_rand, **5**, X\_over, y\_over)

# Using the random search function with

best = [] # creating a empty list

list1 = [] # creating a empty list

**for** i **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times the length of

# the mean\_test\_score

best.append(rand\_search.cv\_results\_["mean\_test\_score"][i]) # appending the best values ot the

# best variable.

list1.append(rand\_search.cv\_results\_["param\_max\_depth"][i]) # appedning the values to the list1.

sorted\_best= [] # creating a empty list

sorted\_list1 = list1.copy() # creating a copy of the list1 variables values.

sorted\_list1.sort() # sorting the KNNs

**for** i **in** range(len(sorted\_list1)): # Looping through the lenght of the sorted\_list1 variable.

**for** j **in** range(len(list1)): # Looping through the lenght of the list1 variable.

**if** sorted\_list1[i] == list1[j]: # checking that if the values of sorted\_list1[i] is equal to list1[j]

sorted\_best.append(best[j]) # appeding the best[j] values to the sorted\_best list.

max\_depth\_list = list(range(**101**,**150**)) # a lsit with values specified.

splitter\_list = ['random'] # a lsit with values specified.

para\_rand = dict(max\_depth = max\_depth\_list, splitter = splitter\_list) # adding the above

# lists in a dictinary.

rand\_search = rand\_search\_fun(DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**, max\_leaf\_nodes = None, \

min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', \

max\_features = None), para\_rand, **5**, X\_over, y\_over)

# Using the random search function with

random = [] # creating a empty list

list2 = [] # creating a empty list

**for** i **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times the length of

# the mean\_test\_score

random.append(rand\_search.cv\_results\_["mean\_test\_score"][i]) # appending the random values ot the

# random variable.

list2.append(rand\_search.cv\_results\_["param\_max\_depth"][i]) # appedning the items to the list2.

sorted\_random = [] # creating a empty list

sorted\_list2 = list2.copy() # creating a copy of the list2 variables values.

sorted\_list2.sort() # sorting the list

**for** i **in** range(len(sorted\_list2)): # Looping through the lenght of the sorted\_list2 variable.

**for** j **in** range(len(list2)): # Looping through the lenght of the list2 variable.

**if** sorted\_list2[i] == list2[j]: # checking that if the values of sorted\_list2[i] equal list2[j]

sorted\_random.append(random[j]) # appeding the random[j] values to the sorted\_random list.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Random Search Hyper parameter Comparison", fontsize=**25**, color="blue") # setting the plot title.

plt.plot(sorted\_list1,sorted\_best , "--", marker = "o", label="splitte best", color="red") # setting the best

plt.plot(sorted\_list2,sorted\_random ,":", marker = "\*", label="splitte random", color="blue") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('max depth',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# X and y obtained by using Corelation matrix

## Random search

min\_samples\_split\_list = list(range(**2**,**50**)) # a lsit with values specified.

splitter\_list = ['best'] # a lsit with values specified.

para\_rand = dict(min\_samples\_split = min\_samples\_split\_list, splitter = splitter\_list) # adding the above

# lists in a dictinary.

rand\_search = rand\_search\_fun(DecisionTreeClassifier(min\_samples\_leaf = **10**, max\_leaf\_nodes = **20**, \

min\_weight\_fraction\_leaf = **0.0**, max\_depth = **3**, criterion = 'entropy', \

max\_features = "sqrt"), para\_rand, **5**, X2, y) # Using the grid search function with

best = [] # creating a empty list

list1 = [] # creating a empty list

**for** i **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times the length of

# the mean\_test\_score

best.append(rand\_search.cv\_results\_["mean\_test\_score"][i]) # appending the best values ot the

# best variable.

list1.append(rand\_search.cv\_results\_["param\_min\_samples\_split"][i]) # appedning the values to the list1.

sorted\_best= [] # creating a empty list

sorted\_list1 = list1.copy() # creating a copy of the list1 variables values.

sorted\_list1.sort() # sorting the KNNs

**for** i **in** range(len(sorted\_list1)): # Looping through the lenght of the sorted\_list1 variable.

**for** j **in** range(len(list1)): # Looping through the lenght of the list1 variable.

**if** sorted\_list1[i] == list1[j]: # checking that if the values of sorted\_list1[i] is equal to list1[j]

sorted\_best.append(best[j]) # appeding the best[j] values to the sorted\_best list.

min\_samples\_split\_list = list(range(**2**,**50**)) # a lsit with values specified.

splitter\_list = ['random'] # a lsit with values specified.

para\_rand = dict(min\_samples\_split = min\_samples\_split\_list, splitter = splitter\_list) # adding the above

# lists in a dictinary.

rand\_search = rand\_search\_fun(DecisionTreeClassifier(min\_samples\_leaf = **10**, max\_leaf\_nodes = **20**, \

min\_weight\_fraction\_leaf = **0.0**, max\_depth = **3**, criterion = 'entropy', \

max\_features = "sqrt"), para\_rand, **5**, X2, y) # Using the grid search function with

random = [] # creating a empty list

list2 = [] # creating a empty list

**for** i **in** range(len(rand\_search.cv\_results\_["mean\_test\_score"])): # looping through the times the length of

# the mean\_test\_score

random.append(rand\_search.cv\_results\_["mean\_test\_score"][i]) # appending the random values ot the

# random variable.

list2.append(rand\_search.cv\_results\_["param\_min\_samples\_split"][i]) # appedning the items to the list2.

sorted\_random = [] # creating a empty list

sorted\_list2 = list2.copy() # creating a copy of the list2 variables values.

sorted\_list2.sort() # sorting the list

**for** i **in** range(len(sorted\_list2)): # Looping through the lenght of the sorted\_list2 variable.

**for** j **in** range(len(list2)): # Looping through the lenght of the list2 variable.

**if** sorted\_list2[i] == list2[j]: # checking that if the values of sorted\_list2[i] equal list2[j]

sorted\_random.append(random[j]) # appeding the random[j] values to the sorted\_random list.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Random Search Hyper parameter Comparison", fontsize=**25**, color="blue") # setting the plot title.

plt.plot(sorted\_list1,sorted\_best , "--", marker = "o", label="splitte best", color="red") # setting the best

plt.plot(sorted\_list2,sorted\_random ,":", marker = "\*", label="splitte random", color="blue") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('min samples split',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# X and y un\_edited

## Grid Search

max\_leaf\_nodes\_list = list(range(**1**,**20**)) # a list with values specified.

min\_samples\_leaf\_list = [**32**, **33**, **34**] # a list with values specified.

min\_samples\_split\_list = [**29**, **31**] # a list with values specified.

**for** i **in** range(len(min\_samples\_split\_list)):

**if** i == **0**:

para\_grid = dict(max\_leaf\_nodes = max\_leaf\_nodes\_list, min\_samples\_leaf = min\_samples\_leaf\_list, \

min\_samples\_split = [min\_samples\_split\_list[i]]) # adding the above

# lists in a dictinary.

grid\_search = Grid\_search\_fun(DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**, max\_depth = **4**, \

criterion = 'entropy', \

max\_features = "auto", splitter = "random"), para\_grid, **5**, X, y) # X and y un\_edited

**if** i == **1**:

para\_grid = dict(max\_leaf\_nodes = max\_leaf\_nodes\_list, min\_samples\_leaf = min\_samples\_leaf\_list, \

min\_samples\_split = [min\_samples\_split\_list[i]]) # adding the above

# lists in a dictinary.

grid\_search2 = Grid\_search\_fun(DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**, max\_depth = **4**, \

criterion = 'entropy', \

max\_features = "auto", splitter = "random"), para\_grid, **5**, X, y) # X and y un\_edited

## Random search # Using the grid search gunction with

list1 = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all even number

# indexes using a for loop.

**if** grid\_search.cv\_results\_["param\_min\_samples\_leaf"][i] == **32**:

list1.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the 32 values to the

# varibale list1.

list2 = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search.cv\_results\_["param\_min\_samples\_leaf"][i] == **33**:

list2.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the 33 values to the

# varibale list2.

list3 = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search.cv\_results\_["param\_min\_samples\_leaf"][i] == **34**:

list3.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the 34 values to the

# varibale lst3.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Grid Search Hyper parameter Comparison with 29 min samples split", fontsize=**25**, color="blue") # setting the

# plot title.

plt.plot(max\_leaf\_nodes\_list,list1 , "--", marker = "o", label="min samples leaf 32", color="red") # setting the best

plt.plot(max\_leaf\_nodes\_list,list2 ,":", marker = "\*", label="min samples leaf 33", color="blue") # setting the random

plt.plot(max\_leaf\_nodes\_list,list3 ,":", marker = "d", label="min samples leaf 34", color="orange") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('max leaf nodes',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

list1 = [] # a empty list is created

**for** i **in** range(len(grid\_search2.cv\_results\_["mean\_test\_score"])): # getting all even number

# indexes using a for loop.

**if** grid\_search2.cv\_results\_["param\_min\_samples\_leaf"][i] == **32**:

list1.append(grid\_search2.cv\_results\_["mean\_test\_score"][i]) # appending the 32 values to the

# varibale list1.

list2 = [] # a empty list is created

**for** i **in** range(len(grid\_search2.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search2.cv\_results\_["param\_min\_samples\_leaf"][i] == **33**:

list2.append(grid\_search2.cv\_results\_["mean\_test\_score"][i]) # appending the 33 values to the

# varibale list2.

list3 = [] # a empty list is created

**for** i **in** range(len(grid\_search2.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search2.cv\_results\_["param\_min\_samples\_leaf"][i] == **34**:

list3.append(grid\_search2.cv\_results\_["mean\_test\_score"][i]) # appending the 34 values to the

# varibale list3.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Grid Search Hyper parameter Comparison with 31 min samples split", fontsize=**25**, color="blue") # setting the

# plot title.

plt.plot(max\_leaf\_nodes\_list,list1 , "--", marker = "o", label="min samples leaf 32", color="red") # setting the best

plt.plot(max\_leaf\_nodes\_list,list2 ,":", marker = "\*", label="min samples leaf 33", color="blue") # setting the random

plt.plot(max\_leaf\_nodes\_list,list3 ,":", marker = "d", label="min samples leaf 34", color="orange") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('max leaf nodes',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# Over Sampled X and y values

## Grid Search

max\_depth\_list = list(range(**101**,**150**)) # a list with values specified.

splitter\_list = ['best', 'random'] # a list with values specified.

para\_grid = dict(max\_depth = max\_depth\_list, splitter = splitter\_list) # adding the above

# lists in a dictinary.

grid\_search = Grid\_search\_fun(DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**, max\_leaf\_nodes = None, \

min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', \

max\_features = None), para\_grid, **5**, X\_over, y\_over) # Using the grid search gunction with

best = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all even number

# indexes using a for loop.

**if** grid\_search.cv\_results\_["param\_splitter"][i] == "best":

best.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the best values to the

# varibale best.

random = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search.cv\_results\_["param\_splitter"][i] == "random":

random.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the random values to the

# varibale random.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Grid Search Hyper parameter Comparison with ", fontsize=**25**, color="blue") # setting the plot title.

plt.plot(max\_depth\_list,best , "--", marker = "o", label="splitter best", color="red") # setting the best

plt.plot(max\_depth\_list,random ,":", marker = "\*", label="splitter random", color="blue") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('Max depth',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# X and y obtained by using Corelation matrix

## Grid Search

min\_samples\_split\_list = list(range(**2**,**50**)) # a list with values specified.

splitter\_list = ['best', 'random'] # a list with values specified.

para\_grid = dict(min\_samples\_split = min\_samples\_split\_list, splitter = splitter\_list) # adding the above

# lists in a dictinary.

grid\_search = Grid\_search\_fun(DecisionTreeClassifier(min\_samples\_leaf = **10**, max\_leaf\_nodes = **20**, \

min\_weight\_fraction\_leaf = **0.0**, max\_depth = **3**, criterion = 'entropy', \

max\_features = "sqrt"), para\_grid, **5**, X2, y) # Using the grid search gunction with

best = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all even number

# indexes using a for loop.

**if** grid\_search.cv\_results\_["param\_splitter"][i] == "best":

best.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the best values to the

# varibale best.

random = [] # a empty list is created

**for** i **in** range(len(grid\_search.cv\_results\_["mean\_test\_score"])): # getting all odd number indexes

# using a for loop.

**if** grid\_search.cv\_results\_["param\_splitter"][i] == "random":

random.append(grid\_search.cv\_results\_["mean\_test\_score"][i]) # appending the random values to the

# varibale random.

plt.figure(dpi=**100**, figsize=(**14**, **7**)) # setting the plot size

plt.title("Grid Search Hyper parameter Comparison with ", fontsize=**25**, color="blue") # setting the plot title.

plt.plot(min\_samples\_split\_list,best , "--", marker = "o", label="splitter best", color="red") # setting the best

plt.plot(min\_samples\_split\_list,random ,":", marker = "\*", label="splitter random", color="blue") # setting the random

# variables values to the plot on the y axis and max depth on the x axis.

plt.xlabel('min samples split',fontsize=**20**) # Setting the x label.

plt.ylabel("Avarage CV Score",fontsize=**20**) # setting the y label.

plt.legend(loc='upper left') # setting the legend.

plt.grid() # setting a grid

# X and y un\_edited

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

Classifier\_function(DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**, max\_depth = **4**, criterion = 'entropy', \

max\_features = "auto", splitter = "random", max\_leaf\_nodes = **6**, \

min\_samples\_leaf = **33**, min\_samples\_split = **31**), X\_train, y\_train,X\_test,\

y\_test, "Decision Tree")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Results after feature scaling

# StandardScaler

featurescaling(StandardScaler(), X\_train, X\_test, y\_train, y\_test,DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**,\

max\_depth = **4**, criterion = 'entropy', max\_features = "auto",\

splitter = "random", max\_leaf\_nodes = **6**,min\_samples\_leaf = **33**,\

min\_samples\_split = **31**) ,"Decision Tree")

# performs standard scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# MinMaxScaler

featurescaling(MinMaxScaler(), X\_train, X\_test, y\_train, y\_test, DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**,\

max\_depth = **4**, criterion = 'entropy', max\_features = "auto",\

splitter = "random", max\_leaf\_nodes = **6**,min\_samples\_leaf = **33**,\

min\_samples\_split = **31**) ,"Decision Tree")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# RobustScaler

featurescaling(RobustScaler(), X\_train, X\_test, y\_train, y\_test,DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**,\

max\_depth = **4**, criterion = 'entropy', max\_features = "auto",\

splitter = "random", max\_leaf\_nodes = **6**,min\_samples\_leaf = **33**,\

min\_samples\_split = **31**) ,"Decision Tree")

# performs robust scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Normalizer

featurescaling(Normalizer(), X\_train, X\_test, y\_train, y\_test, DecisionTreeClassifier(min\_weight\_fraction\_leaf = **0.0**,\

max\_depth = **4**, criterion = 'entropy', max\_features = "auto",\

splitter = "random", max\_leaf\_nodes = **6**,min\_samples\_leaf = **33**,\

min\_samples\_split = **31**) ,"Decision Tree")

# Performs normalization scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Over Sampled X and y values

# Without any model Tuning

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

Classifier\_function(DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**, max\_leaf\_nodes = None, \

min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random'), X\_train\_over, y\_train\_over, X\_test\_over,\

y\_test\_over, "Decision Tree")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Feature scaling

# StandardScaler

featurescaling(StandardScaler(), X\_train\_over, X\_test\_over, y\_train\_over, \

y\_test\_over,DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**,\

max\_leaf\_nodes = None, min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random'), "Decision Tree")

# performs standard scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# MinMaxScaler

featurescaling(MinMaxScaler(), X\_train\_over, X\_test\_over, y\_train\_over,\

y\_test\_over,DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**,\

max\_leaf\_nodes = None, min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random') ,"Decision Tree")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# RobustScaler

featurescaling(RobustScaler(), X\_train\_over, X\_test\_over, y\_train\_over,\

y\_test\_over,DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**,\

max\_leaf\_nodes = None, min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random') ,"Decision Tree")

# performs robust scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Normalizer

featurescaling(Normalizer(), X\_train\_over, X\_test\_over, y\_train\_over,\

y\_test\_over,DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**,\

max\_leaf\_nodes = None, min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random') ,"Decision Tree")

# Performs normalization scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# X and y obtained by using Corelation matrix

# Without any model Tuning

## Checking predicted/actual results

## Checking testing and traning scores

## Checking Actual values classified correctly and wrongly.

## Checking accuracy, precision, recall and f1 scores

Classifier\_function(DecisionTreeClassifier(min\_samples\_leaf = **10**, max\_leaf\_nodes = **20**, min\_weight\_fraction\_leaf = **0.0**,\

max\_depth = **3**, criterion = 'entropy',max\_features = "sqrt",\

min\_samples\_split = **2**, splitter = 'best'), X2\_train, y2\_train,X2\_test, \

y2\_test, "Decision Tree")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Results

# Original Results without hyperparameter Improvement

Classifier\_function(DecisionTreeClassifier(), X\_train\_over, y\_train\_over, X\_test\_over, y\_test\_over, "Decision Tree")

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Best Score and model

# Over Sampled X and y values

# MinMaxScaler

# Results with hyparameters improved

featurescaling(MinMaxScaler(), X\_train\_over, X\_test\_over, y\_train\_over,\

y\_test\_over,DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**,\

max\_leaf\_nodes = None, min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random') ,"Decision Tree")

# performs Minmax scaling

# Performs traing, testing prediction.

# performs precision, recall, f1-score and support prediction

# plots a confusion matrix

# returns the traing, testing, precision, recall, f1-score

# Un\_edited full model evaluation

crossvalscore(DecisionTreeClassifier(), X\_over, y\_over, **10**) # Performs cross validation.

# Improved model full Evaluation with feature scaling

crossvalscore(DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**,\

max\_leaf\_nodes = None, min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random'), X\_over, y\_over, **10**) # Performs cross validation.

# Overall results and full evaluation

**def** **plot\_graph**(list\_for\_plot, list\_for\_plot2,Accuracy\_type, colour): # funstion with given parameters.

"""

The function plot\_graph plots a lineargraph.

The first parameter list\_for\_plot is the x axis.

The second parameter list\_for\_plot2 is the y axis.

The third parameter Accuracy\_type takes a string.

for example "F1 score"

The fourth parameter colour is the color for the bar plot.

"""

plt.figure() # plots figure

plt.title("Model Evaluation",fontsize=**18**) # Displays plot title

plt.plot(list\_for\_plot, list\_for\_plot2, color = colour) # Displays description of the plots x and y labels.

plt.xlabel("Models") # Displays the x axis for the plot

plt.ylabel(Accuracy\_type) # Displays the y axis for the plot.

plt.grid() # adds a gird to the plot

## k-fold cross-validation

**def** **crossvalscore**(model, X, y, cv\_val): # function to perform cross validation with model X, y and cv\_val as parameters

"""

The crossvalscore function finds the avarage cross validation accuracy, precison, recall, f1 score, and area uner the curve

of a model and its standard deveation.

The first parameter model is the type of model to perform the cross validation on.

The second parameter X is the features.

The third parameter y is the labels.

the foruth parameter cv\_val is the number of times to cross validate the given model.

The crossvalscore function returns the cross validation accuracy, precison, recall, f1 scoer, and area uner the curve

"""

sc = MinMaxScaler() # creating an instance of the object.

X[:, **55**:] = sc.fit\_transform(X[:, **55**:]) # Scaling x\_train

X[:,**55**:] = sc.transform(X[:, **55**:]) # Scaling y\_train

scoring = ['accuracy', 'precision\_macro', 'recall\_macro', 'f1\_macro', 'roc\_auc\_ovr'] # a list with values given.

results = [] # a empty list

results\_std = []

**for** i **in** range(len(scoring)): # looping to the leghth of the variable scoring

**print**("score metric = ", scoring[i]) # printing output

score = cross\_val\_score(estimator = model, X = X, y = y,scoring=scoring[i], cv = cv\_val) # performs different tests to get best accurecy.

**print**("Score : ",score.mean()) # accuracy printed.

**print**("Standard Deviation", score.std()) # standard deveation printed (std -avarage or std+ avarage )

results.append(score.mean())

results\_std.append(score.std())

**print**("**\n**")

**return** results[**0**], results\_std[**0**], results[**1**],results\_std[**1**] ,results[**2**],results\_std[**2**],\

results[**3**],results\_std[**3**], results[**4**], results\_std[**4**]

# Decision Tree

DT\_accuracy,DT\_accuracy\_std, DT\_precision, DT\_precision\_std, DT\_recall,DT\_recall\_std, DT\_f1, DT\_f1\_std, DT\_auc, DT\_auc\_std = crossvalscore(DecisionTreeClassifier(min\_samples\_leaf = **1**, min\_samples\_split = **2**,\

max\_leaf\_nodes = None, min\_weight\_fraction\_leaf = **0.0**,criterion = 'entropy', max\_features = None, \

max\_depth = **121**, splitter = 'random'), X\_over, y\_over, **10**) # Performs cross validation.

# Random Forest

RF\_accuracy,RF\_accuracy\_std, RF\_precision, RF\_precision\_std, RF\_recall,RF\_recall\_std, RF\_f1, RF\_f1\_std, RF\_auc, RF\_auc\_std = crossvalscore(RandomForestClassifier(n\_estimators =**90**,min\_samples\_split = **2**, min\_samples\_leaf = **1**,\

max\_depth = **50**,bootstrap = False, max\_features = 'sqrt', criterion = 'entropy'), X\_over, y\_over, **10**)

# Performs cross validation.

# KNN

KNN\_accuracy,KNN\_accuracy\_std, KNN\_precision, KNN\_precision\_std, KNN\_recall,KNN\_recall\_std, KNN\_f1, KNN\_f1\_std, KNN\_auc, KNN\_auc\_std = crossvalscore(KNeighborsClassifier(n\_neighbors = **15**, weights ='distance'), X\_over, y\_over, **10**) # Performs cross validation.

model\_name = ["KNN","Decision Tree", "Random Forest"] # a list of models used.

Sampling = ["Over Sampled", "Over Sampled", "Over Sampled"] # a list of sampling techniwues used.

scalars = ["MInMax Scalar", "MinMax Scalar", "MinMax Scalar"] # a list of types of scaling techniques used.

mean\_accuracy = [KNN\_accuracy,DT\_accuracy,RF\_accuracy] # a list of model train accuracy

standard\_deaviation = [KNN\_accuracy\_std,DT\_accuracy\_std,RF\_accuracy\_std] # a list of model train standard deveation.

mean\_precision = [KNN\_precision,DT\_precision,RF\_precision] # a list of model precision score

precision\_standard\_deaviation = [KNN\_precision\_std,DT\_precision\_std,RF\_precision\_std] # a list of model precision standard deveation.

mean\_recall = [KNN\_recall,DT\_recall,RF\_recall] # a list of model recall score

recall\_standard\_deaviation = [KNN\_recall\_std,DT\_recall\_std,RF\_recall\_std] # a list of model recall standard deveation.

mean\_f1\_score = [KNN\_f1,DT\_f1,RF\_f1] # a list of model f1 score

f1\_score\_standard\_deaviation = [KNN\_f1\_std,DT\_f1\_std,RF\_f1\_std] # a list of model f1 score standard deveation.

mean\_Area\_under\_curve = [KNN\_auc,DT\_auc,RF\_auc] # a list of model area under curve score

Area\_under\_curve\_standard\_deaviation = [KNN\_auc\_std,DT\_auc\_std,RF\_auc\_std] # a list of model area under curve standard deveation.

Evaluation = pd.DataFrame({

'Model':model\_name,

'Sampling': Sampling,

'scalar type':scalars,

'Mean\_accuracy':mean\_accuracy,

'Mean Accuracy Standard Deaviation':standard\_deaviation ,

'Mean Precision':mean\_precision,

'Precision Standard Deaviation':precision\_standard\_deaviation ,

'Mean Recall':mean\_recall,

'Recall Standard Deaviation':recall\_standard\_deaviation ,

'Mean f1 score':mean\_f1\_score,

'f1 score Standard Deaviation':f1\_score\_standard\_deaviation ,

'Mean Area under curve':mean\_Area\_under\_curve,

'Area under curve Standard Deaviation':Area\_under\_curve\_standard\_deaviation

}) # using lists to make a dataframe

Evaluation # viewing dataframe information

Evaluation.sort\_values(by='Mean f1 score', ascending=False) # viewing dataframe information by f1 score in decending order.

Evaluation.sort\_values(by='Mean Recall', ascending=False) # viewing dataframe information by recall in decending order.

Evaluation.sort\_values(by='Mean Area under curve', ascending=False) # viewing dataframe information by area under the curve

# in decending order.

plot\_bar\_graph(model\_name, mean\_f1\_score, "f1 score", "red") # functions argents given. plots output using given data.

plot\_bar\_graph(model\_name, mean\_recall, "Recall", "blue") # functions argents given. plots output using given data.

plot\_graph(model\_name, mean\_Area\_under\_curve, "Area Under Curve", "black") # functions argents given. plots output

# using given data.

plot\_graph(model\_name, mean\_f1\_score, "f1\_score", "black") # functions argents given. plots output using given data.

plot\_graph(model\_name, mean\_recall, "recall", "black") # functions argents given. plots output using given data.

>

# Appendix C

## Outlier analysis

The outliers in this dataset are present in the Population, Total Labour force, GDPpyear, GDPpcapita, Ref ugee\_population\_by\_country\_or\_territory\_of\_origin and Suicidesper100k columns. The data in these columns is taken from 48 countries in years 1985 to 2016 from different age groups and genders. As seen by Figure 1 Countries that Mongolia has only 10 samples while Mexico has 372 samples and looking at Figure 3 Population it is observed that the population of males aged 75+ is 36300 and looking at Figure 2 Population the average population is 3035752.0 there is a 195.274% difference between these two numbers. Judging by these numbers alone it can be seen that there would be outliers present in the data set.

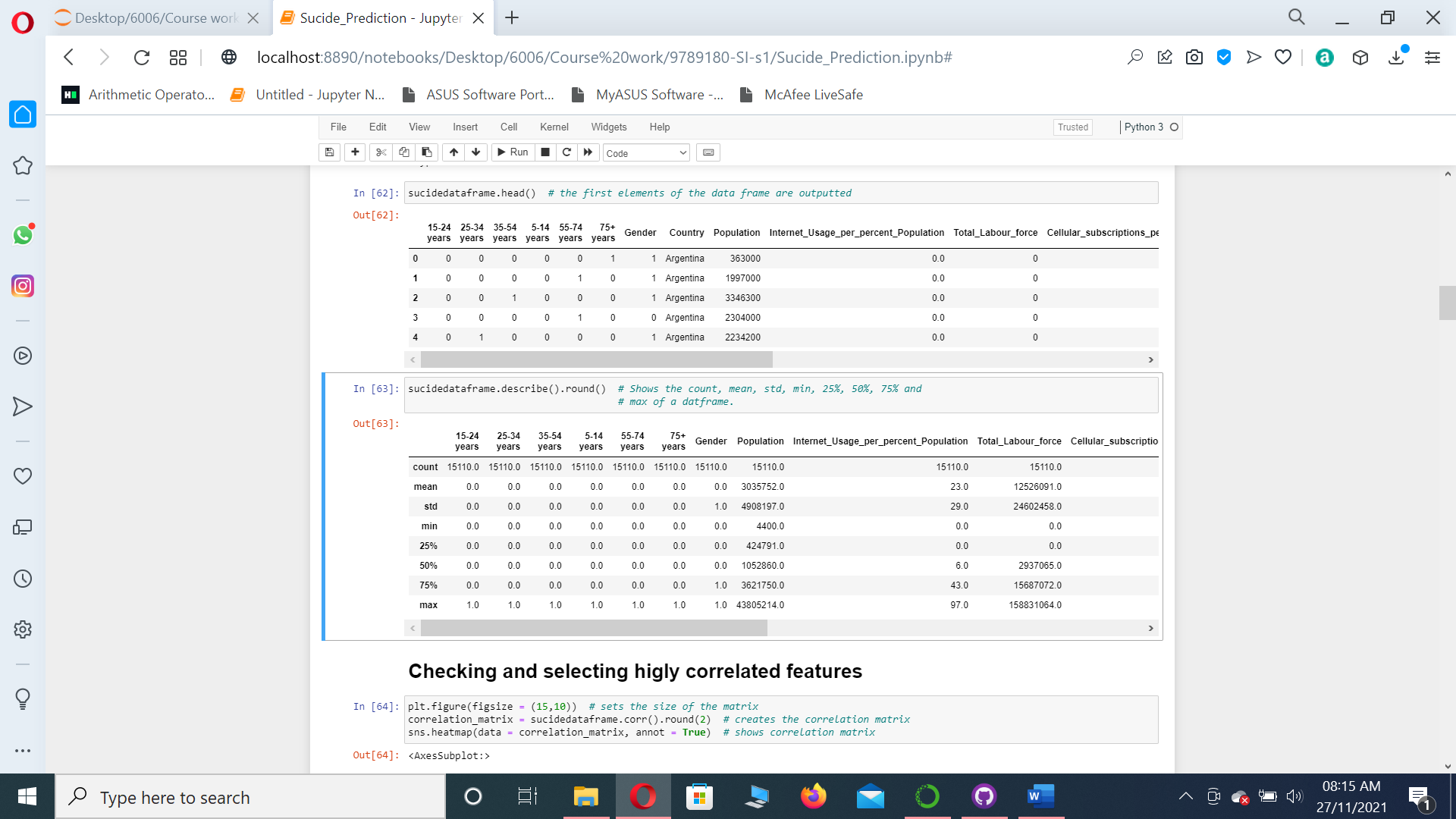


Figure Population

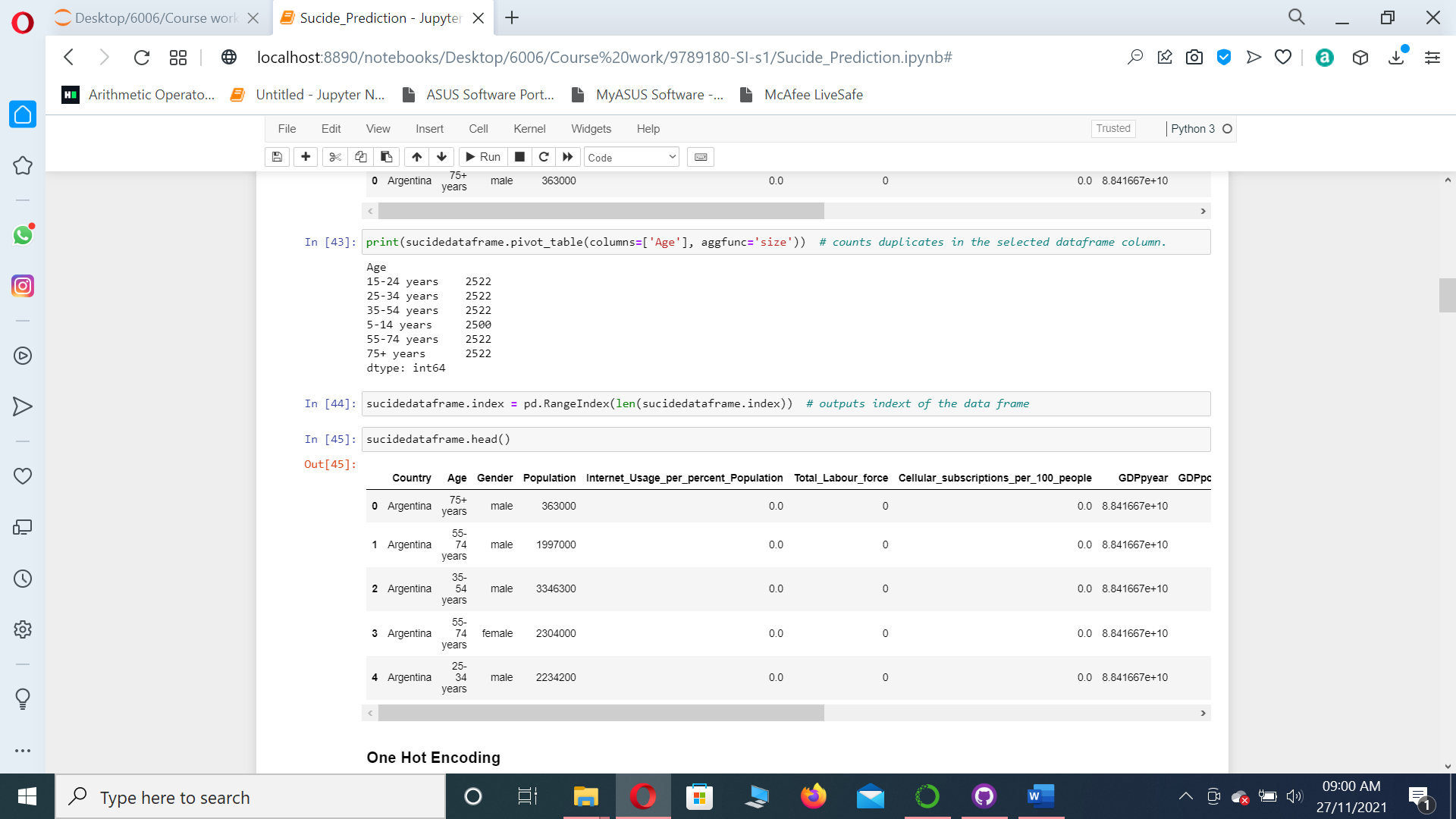


Figure Population

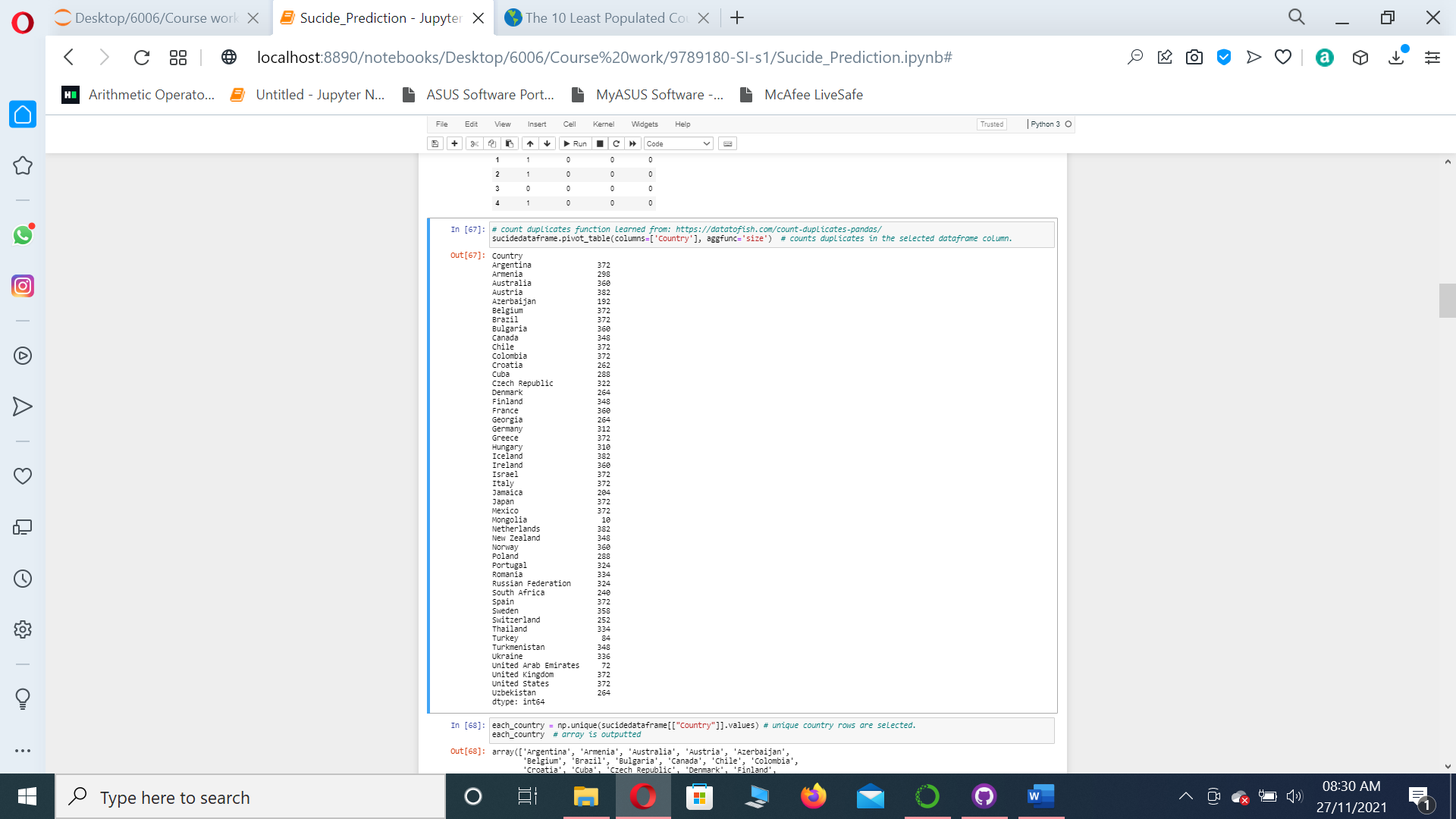
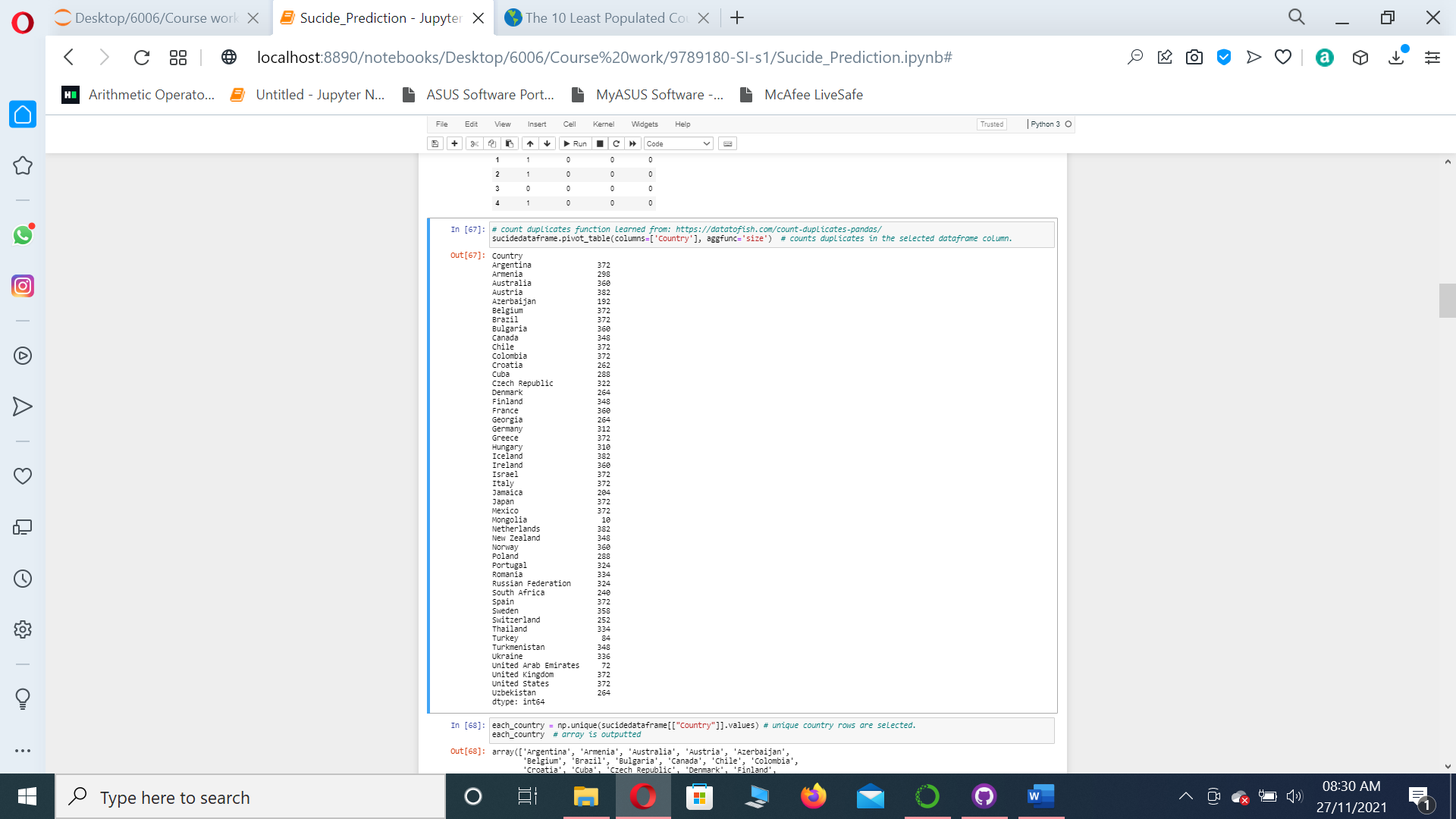


Figure Countries

Now as seen on Figure 4 population by age there is a marginal difference between people aged between 5 – 14 and 13 – 24 from all other aged groups. Which can also indicate outliers but the information does add value to the data analysis.

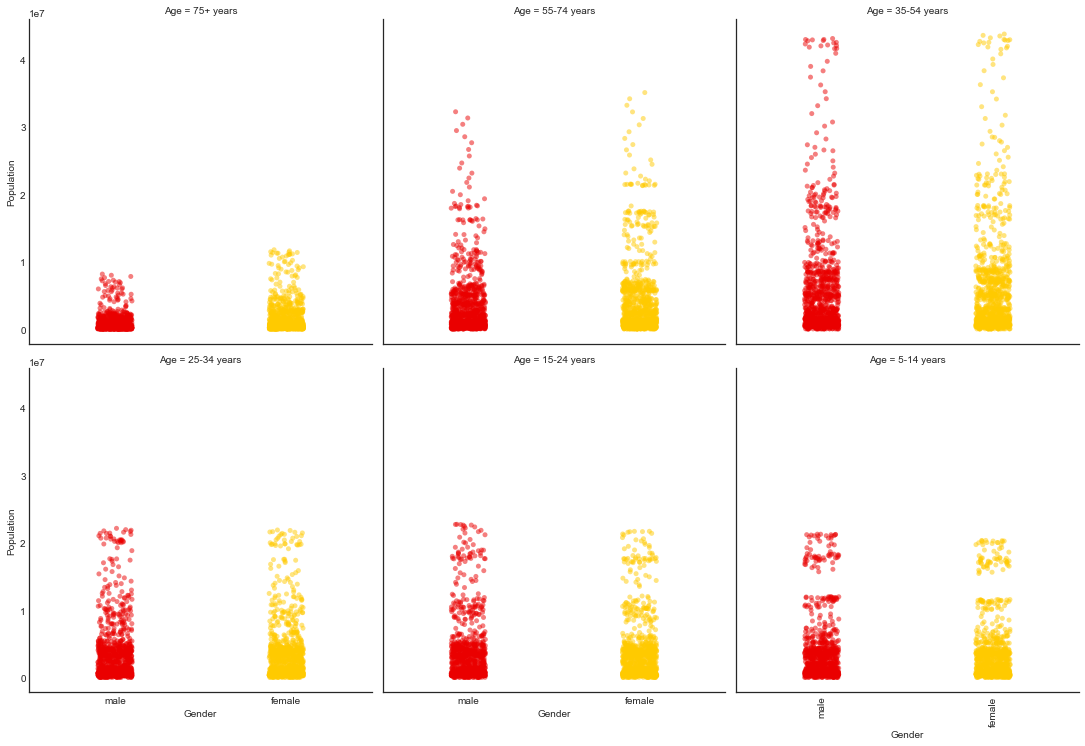


Figure population by age

Observing Figure 5 Suicide by population a observation can be made that males aged 75 + have the highest suicide and males aged 35 – 54 have the highest suicide rate but with a increase in population over 20000000 the suicide levels stop increasing and become steady. Indicating the presence of outliers. which is the same case for people aged 55 – 74.

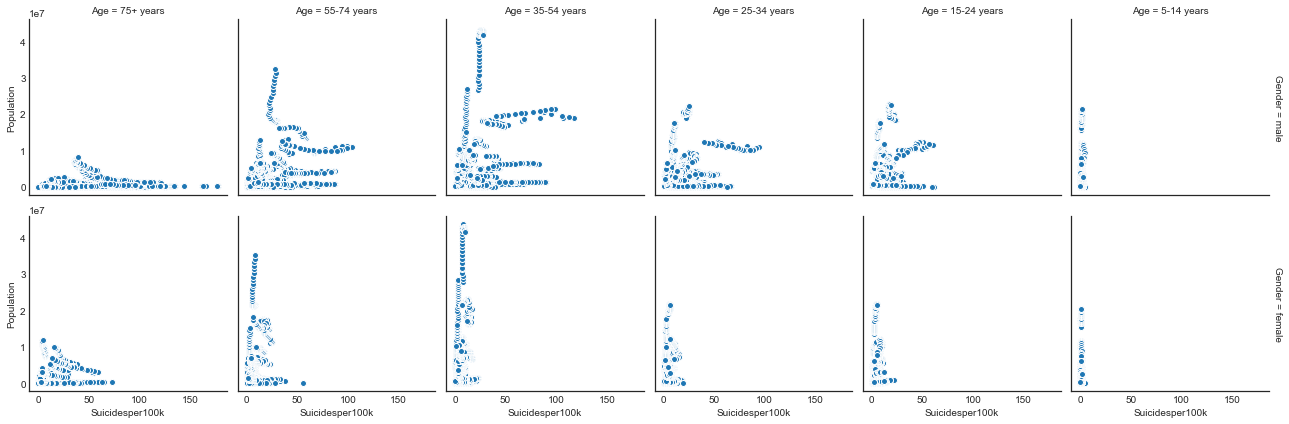


Figure Suicide by population

Observing Figure 6 Refugee\_population\_by\_country\_or\_territory\_of\_origin the rate of sucide in counties with lower refugees is higher as compared to countries with more refugees for males.

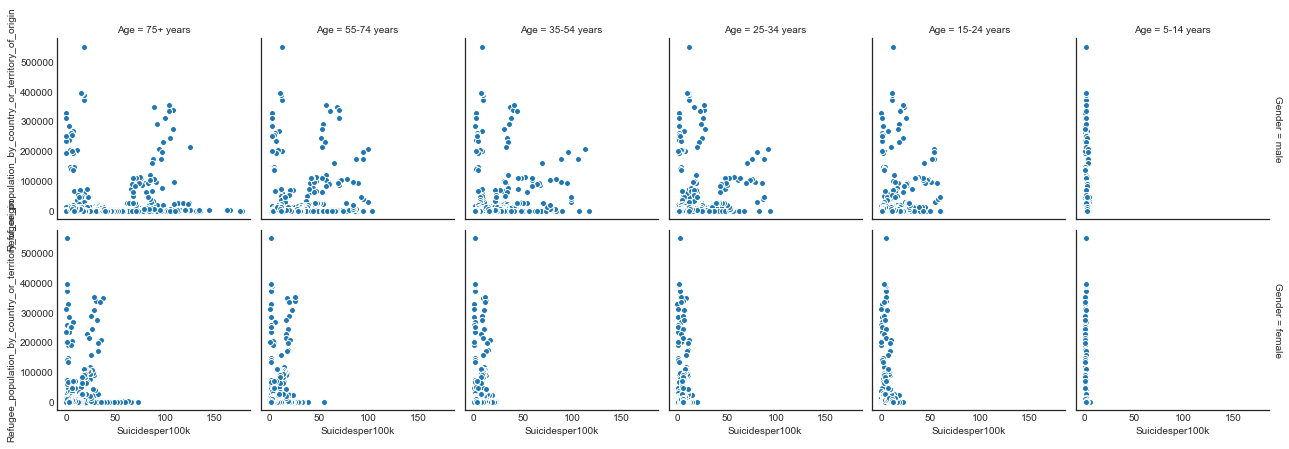


Figure Refugee\_population\_by\_country\_or\_territory\_of\_origin by suicide

Looking at figure Figure 7 GDP per year by suicide it is seen that the number of suicides decrease as the GDP per year of a county increases.

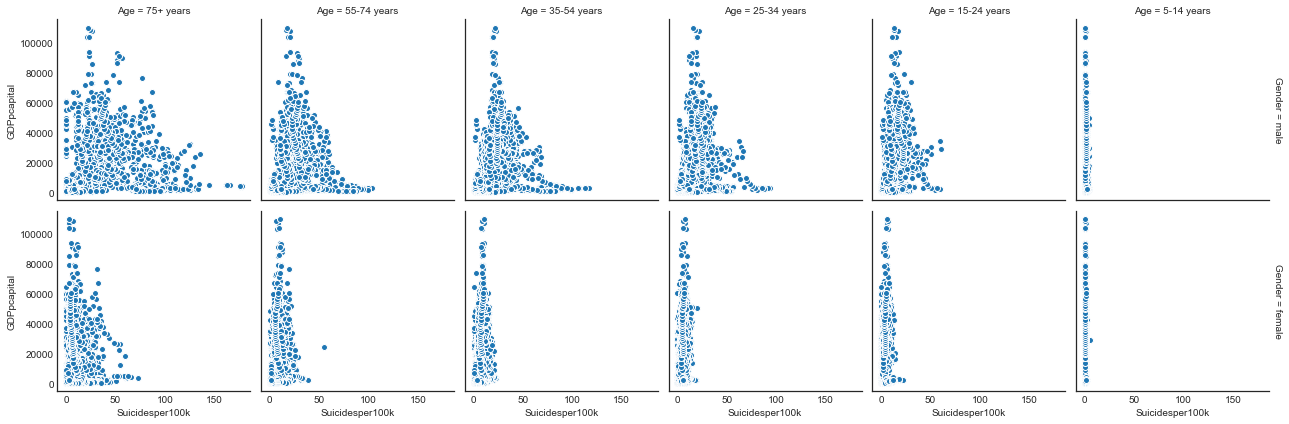


Figure GDP per year by suicide

Although the data is from different counties and different age groups a trend is seen that countries with a higher population and higher refugees the counties also have a higher GDP and so it makes sense that wealthier counties would have more refugees and better quality of life and so lower suicide rate. They are many extreme low values and extreme high values but as observed above the data is from various age ranges , countries and is indicating a pattern so the outliers are providing value to the dataset as they are following clear trend and hence they will be kept for this project.

# Appendix D

## Suicide Label Classification

The maximum number of suicides being 178 see Figure 2 Suicide rate assessment were divided by 2 then rounded and 89 was obtained. Number of suicides between 89 to 178 were decided as maximum chances of suicide and were labelled as 5. Then 89 was again divided by 2 giving the number then rounded giving 44. The number of suicides between 44 to 89 were classified as high-risk suicides and were labelled as 4. The number 44 was again divided by 2 giving the number 22. The rage between 22 and 44 was selected as medium risk suicide and labelled as 3. The number 22 was again divided by 2 giving the number 11. The rage of suicides between 11 and 22 were classified as low risk of suicide and were labelled as 2. Then finally the number of suicides below 11 were classified as minimum risk of suicide and were labelled as 1.

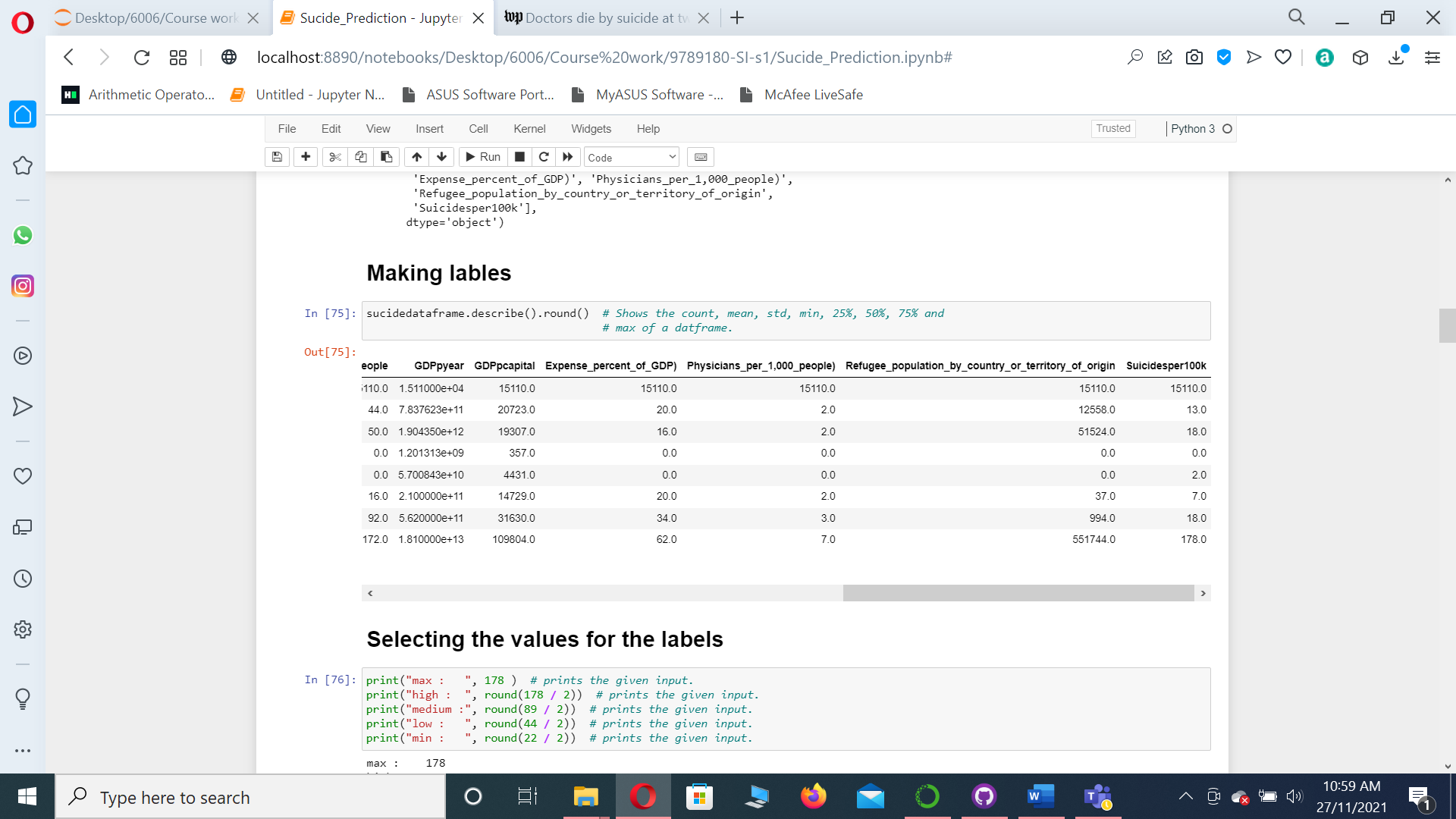
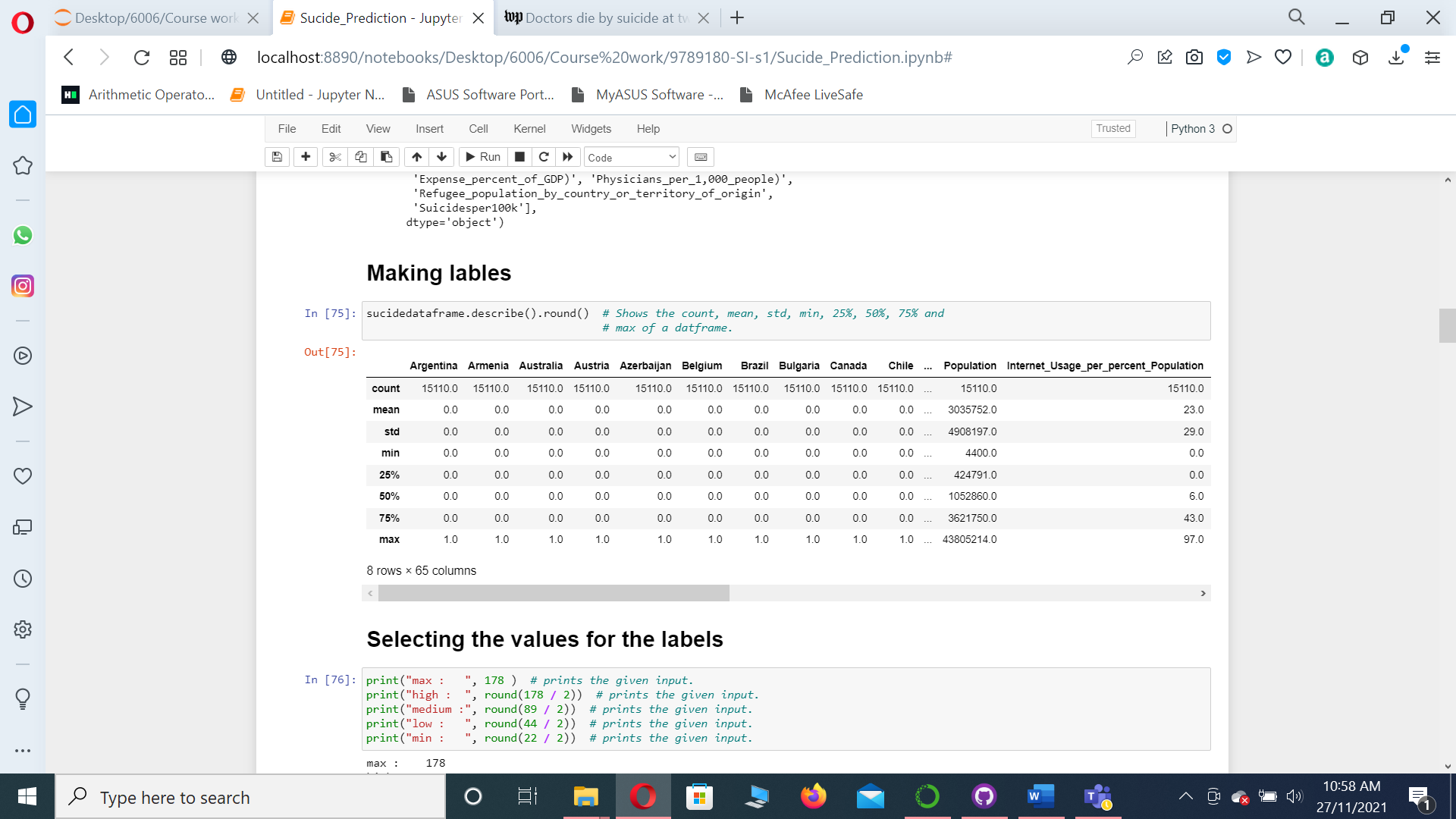


Figure Suicide rate assessment

# Appendix E

## Model optimization

As the models performed using the oversampled best feature the best so only those model optimization techniques will be mentioned here. Other optimization of the default and the set of label and features obtained using confusing matrix can be found online at: <https://github.coventry.ac.uk/iftikhars/9789180-SI-s1>

### KNN

### KNN with hyperparameters optimized using different neighbours and weight set as distance

When preforming cross validation to improve model hyperparameters 1 neighbour with weight distance was determined to give the best results.

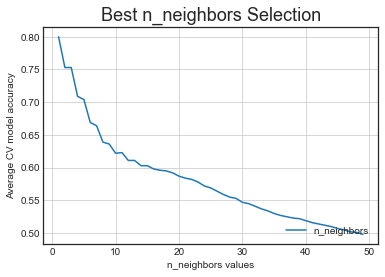


Figure KNN Cross Validation

When preforming Random search to improve model hyperparameters 3 neighbour with weight distance was determined to give the best results.

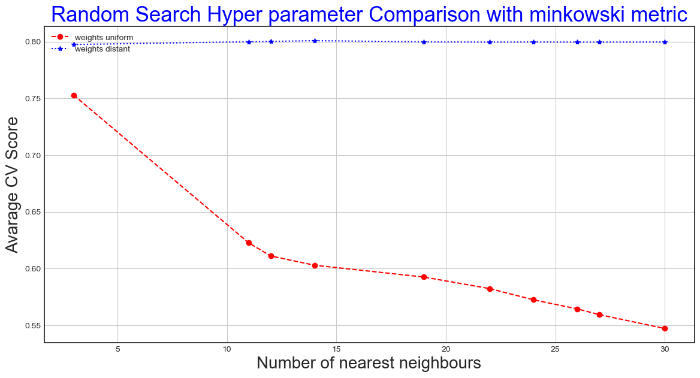


Figure KNN Random Search

When preforming Grid search to improve model hyperparameters 15 neighbours with weight distance was determined to give the best results.

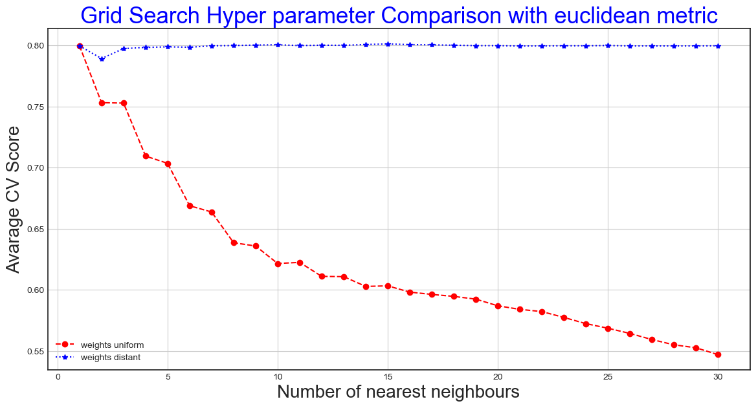


Figure KNN Grid Search

The hyperparameters metrics algorithms and leaf\_size had no effect on the above models performance improvement so they were kept as default.

With 1 neighbour using standard scalar it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9910556787942809 % area under the curve.

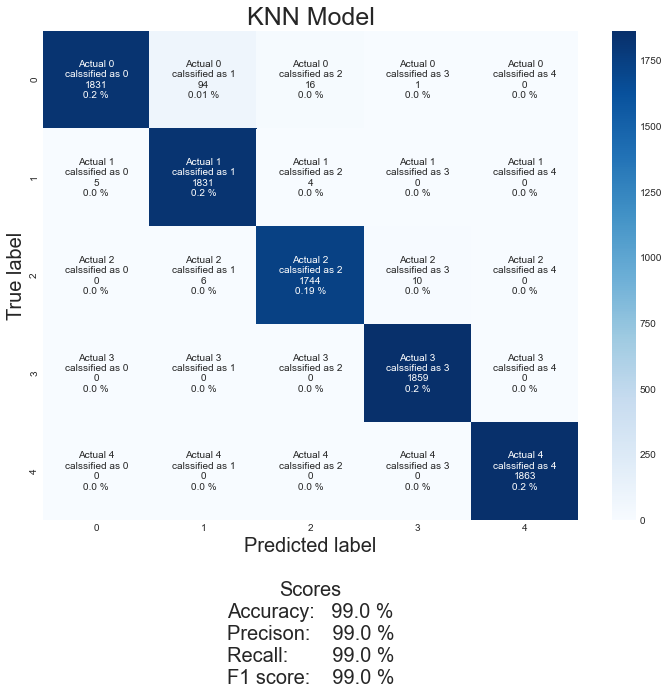


Figure KNN Standardscalar

With 1 neighbour using MinMaxScalar on the features it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9920822625129446 % area under the curve.



Figure KNN MinMax1

With 3 neighbour using MinMaxScalar on the features it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9942384129273425 % area under the curve.

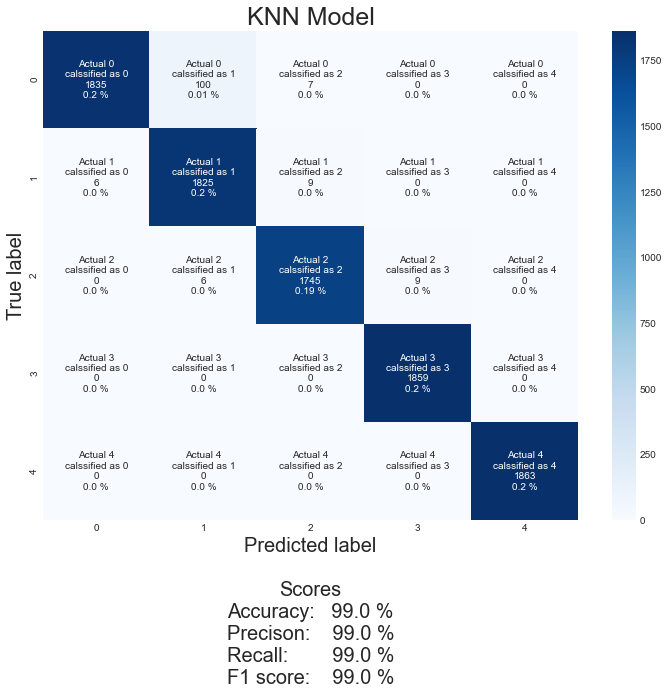


Figure KNN MinMax3

With 1 neighbour using Normalizer on the features it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9921664206004255 % area under the curve.

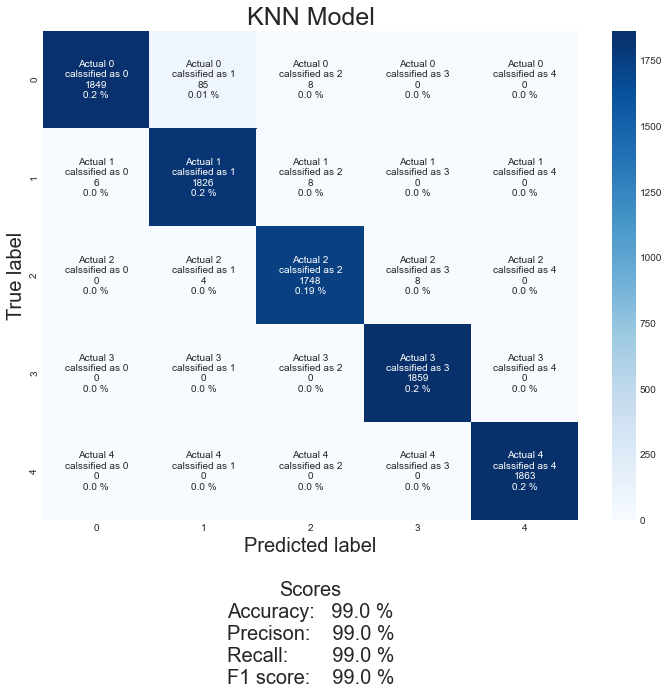


Figure KNN Normalizer

With 3 neighbour using Normalizer it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9941760974789513 % area under the curve.

As it can be seen that With 3 neighbour using MinMaxScalar it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9942384129273425 % area under the curve. But the model is further evaluated with cross validation with the different neighbours to find the best neighbours.

When evaluated with cross validation the on the default training and testing set 71.71 % accuracy and 2.11 % standard deviation were achieved.

When evaluated with cross validation it performed best using 15 neighbours, weight set as distance with 80.22 % accuracy and 1.40 % standard deviation.

When evaluated with cross validation it performed best using 3 neighbours, weight set as distance with 79.96 % accuracy and 1.43 % standard deviation.

When evaluated with cross validation it performed best using 1 neighbour, weight set as distance with 80.28 % accuracy and 2.05 % standard deviation.

When considering the accuracy alone KNN with 1 neighbour performs better but when considering the standard deviation, the 15 neighbours outperform the 1 neighbour and perform better .

15 neighbours will be selected for the KNN model as hyperparameters for the final evaluation with uniform weight.

### Random Forest

#### Random forest with hyperparameters optimized

Random forest n\_estimators hyperparameters were firstly tuned using cross validation in a given range of numbers.

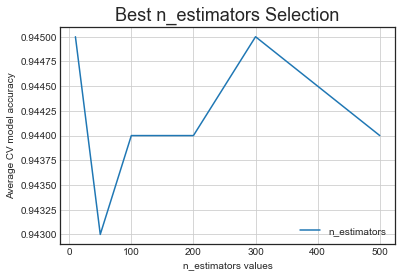


Figure Random Forest Cross Validation 1

The range is narrowed down and the hyperparameter is further improved.

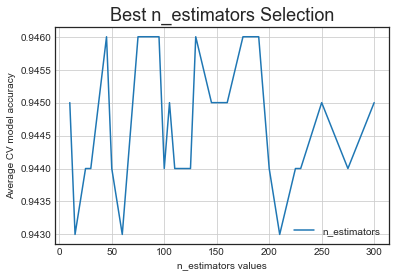


Figure Random Forest Cross Validation 2

Random forest Max\_depth hyperparameters were firstly tuned using cross validation in a given range of numbers.

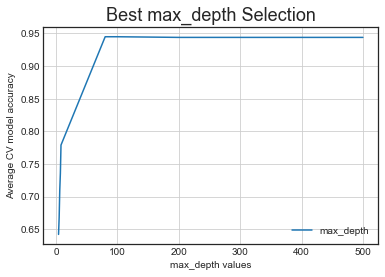


Figure Random Forest Cross Validation 3

The range is narrowed down and the hyperparameter is further improved.



Figure Random Forest Cross Validation 4

As Random forest is computationally very heavy so a lower value of the hyperparameters n\_estimators with max\_depth was perfred to decrease the computation time. Cross validation tests are run using these hyperparameters and it is determined that 90 n\_estimators with max\_depth 50 gives the best results.

1 min\_samples\_leaf was selected as the best hyperparameter using cross validation.

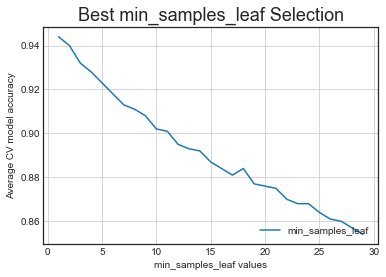


Figure Random Forest Cross Validation 5

2 and 3 min\_samples\_split was selected as the best hyperparameter using cross validation.

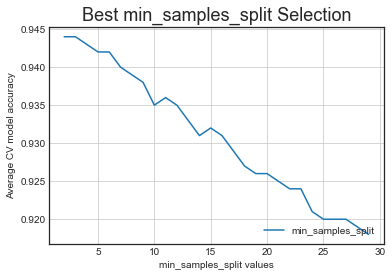


Figure Random Forest Cross Validation 6

The min\_samples\_split were further tested using cross validation and it was determined that 2 min\_samples\_split gave the best results.

The hyperparameters, bootstrap\_list, max\_features\_list and criterion\_list was further optimized using grid search and random search in respected to our selected hyperparameters using cross validation.

After both the searches it was determined that best results are obtained when using 90 n\_estimators, 2 min\_samples\_split, 1 min\_samples\_leaf, 50 max\_depth, False bootstrap, 'sqrt' max\_features and 'entropy' criterion

When using 90 n\_estimators, 2 min\_samples\_split, 1 min\_samples\_leaf, 50 max\_depth, False bootstrap, 'sqrt' max\_features and 'entropy' criterion as hyperparameters on the default and scaled models the results are as follows.

Using the default on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.999694146275993 % area under the curve.

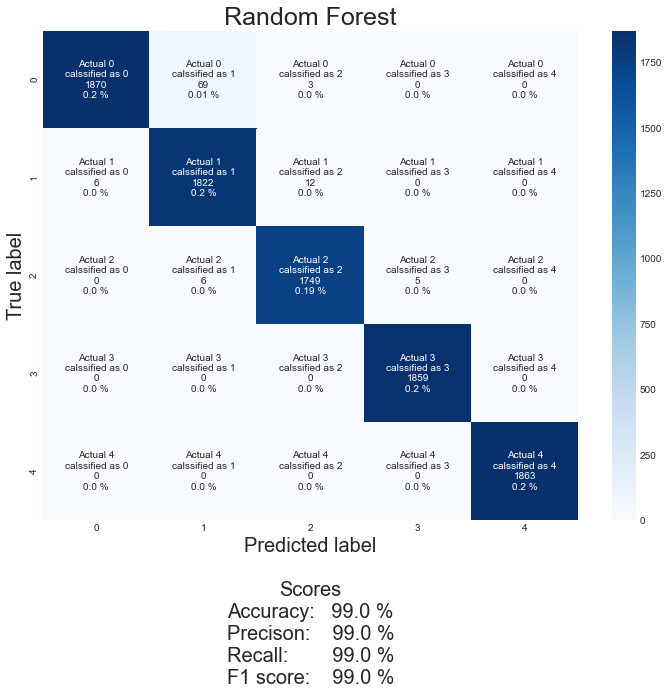


Figure Random forest default

Using the standardscalar on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9996822954596178 % area under the curve.

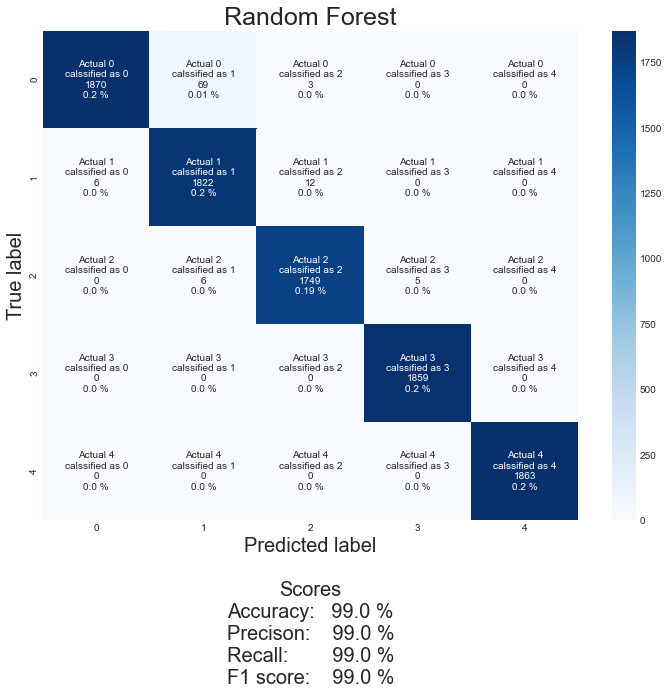


Figure Random forest StandardScalar

Using the MinMaxScalar on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.999747995347186 % area under the curve.

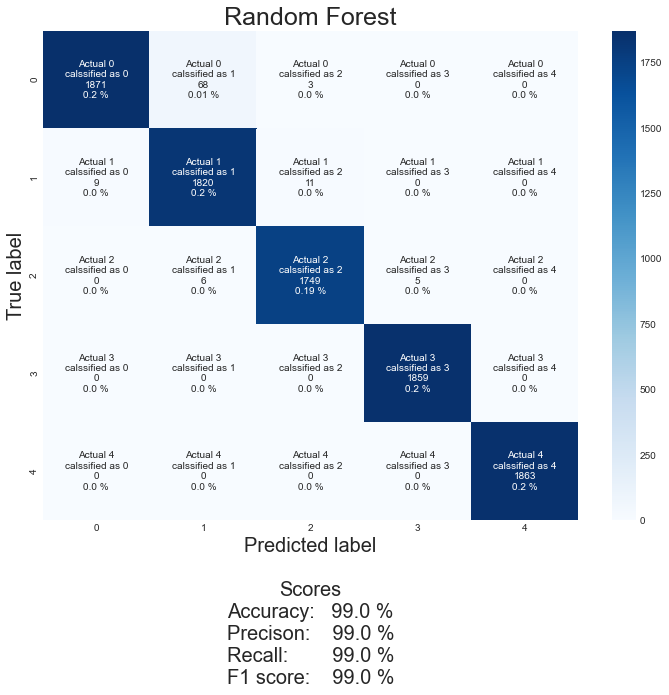


Figure Random forest MinMaxScalar

Using the RobustScalar on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9996979450769103 % area under the curve.



Figure Random forest RobustScalar

Using the Normalizer on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9997492068811304 % area under the curve.

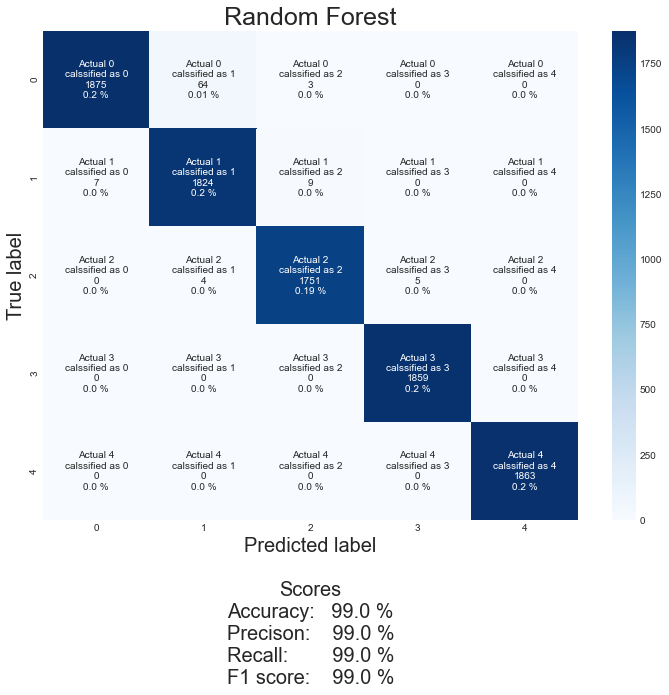


Figure Random forest Normalizer

When using 90 n\_estimators, 2 min\_samples\_split, 1 min\_samples\_leaf, 50 max\_depth, False bootstrap, 'sqrt' max\_features and 'entropy' criterion as hyperparameters all the scaling techniques give , it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with MinMaxScalar has the best area under the curve is achieved so it is selected as the best scaling technique.

After cross validation of these results with the default results.

The defaults model results were 94.79 % with 1.28 % standard deviation.

The results after using minmax scaler with hyperparameter optimization were 94.79 % with 1.16 standard deviation.

### Decision Tree

#### Decision Tree with hyperparameters optimized

Decision Tree max\_depth hyperparameters were tuned using cross validation in a given range of numbers.

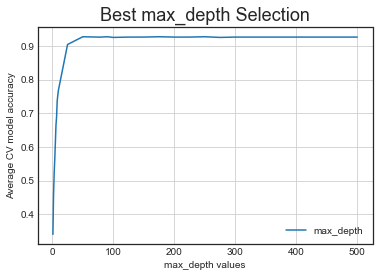


Figure Decision Tree with cross validation 1

Decision Tree min\_samples\_leaf hyperparameters were tuned using cross validation in a given range of numbers.

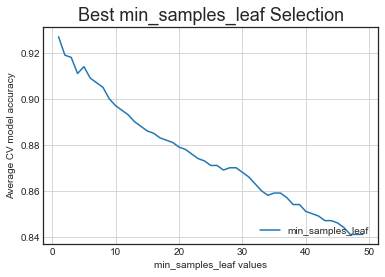


Figure Decision Tree with cross validation 2

Decision Tree min\_samples\_split hyperparameters were tuned using cross validation in a given range of numbers.

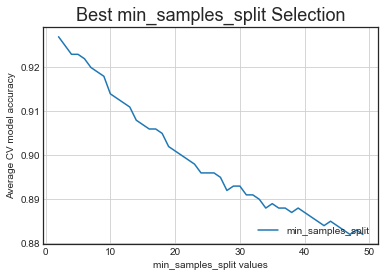


Figure Decision Tree with cross validation 3

Decision Tree min\_weight\_fraction\_leaf hyperparameters were tuned using cross validation in a given range of numbers.



Figure Decision Tree with cross validation 4

Decision Tree max\_leaf\_nodes hyperparameters were tuned using cross validation in a given range of numbers.

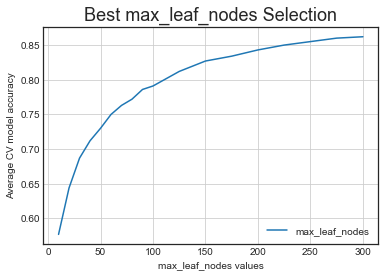


Figure Decision Tree with cross validation 5

Decision Tree criterion hyperparameters were tuned using cross validation in a given range of numbers.

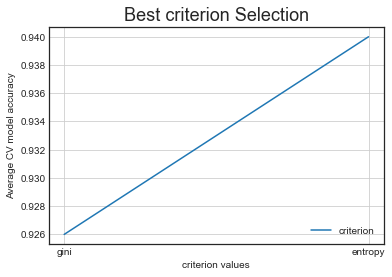


Figure Decision Tree with cross validation 6

Decision Tree max\_features hyperparameters were tuned using cross validation in a given range of numbers.

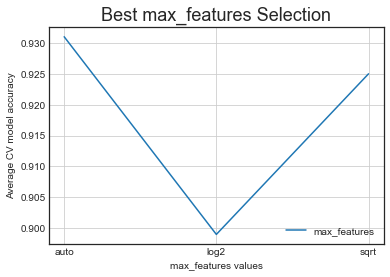


Figure Decision Tree with cross validation 7

Decision Tree splitter hyperparameters were tuned using cross validation in a given range of numbers.

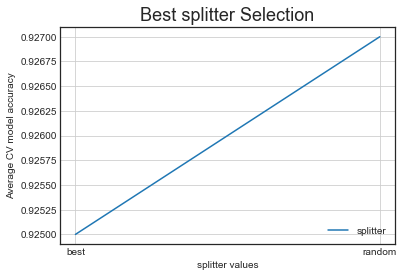


Figure Decision Tree with cross validation 8

After which the hyperparameters max\_depth and splitter further tuned using grid search and random search.

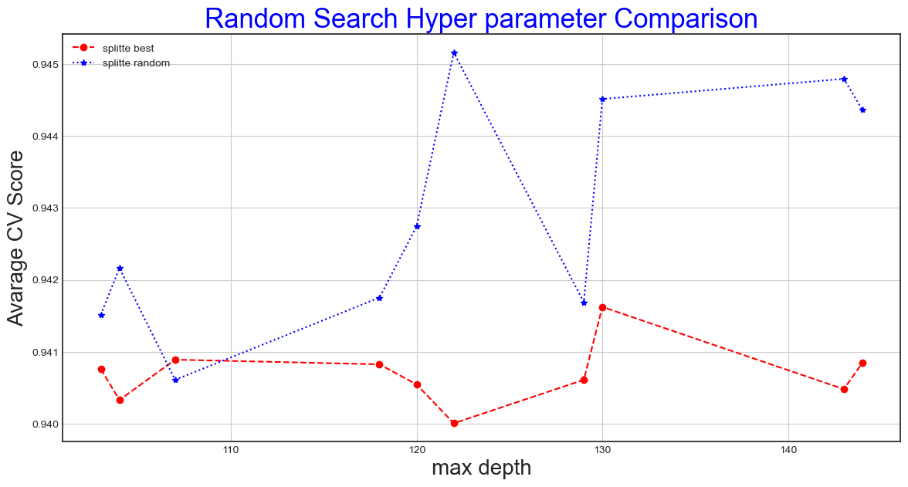


Figure Decision Tree with Random Search

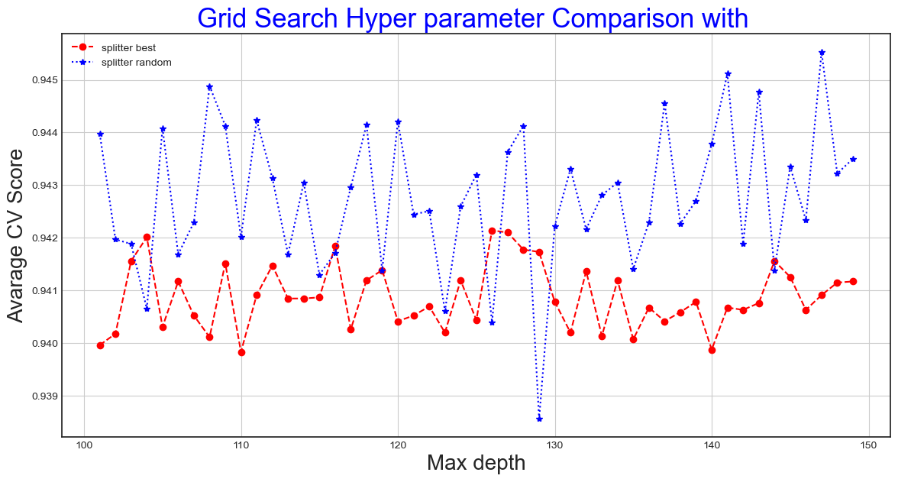


Figure Decision Tree with Grid Search

After both the grid and random search it is concluded that with 1 min\_samples\_leaf ,2 min\_samples\_split , None max\_leaf\_nodes , 0.0 min\_weight\_fraction\_leaf , 'entropy' criterion, None max\_features, 121 max\_depth, 'random' splitter best results are obtained. These hyperparameters are now tested with the default unscaled oversampled features as well as these features after scaling. The results from these tests are as follows.

Using the default features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9923048031499204 % area under the curve.

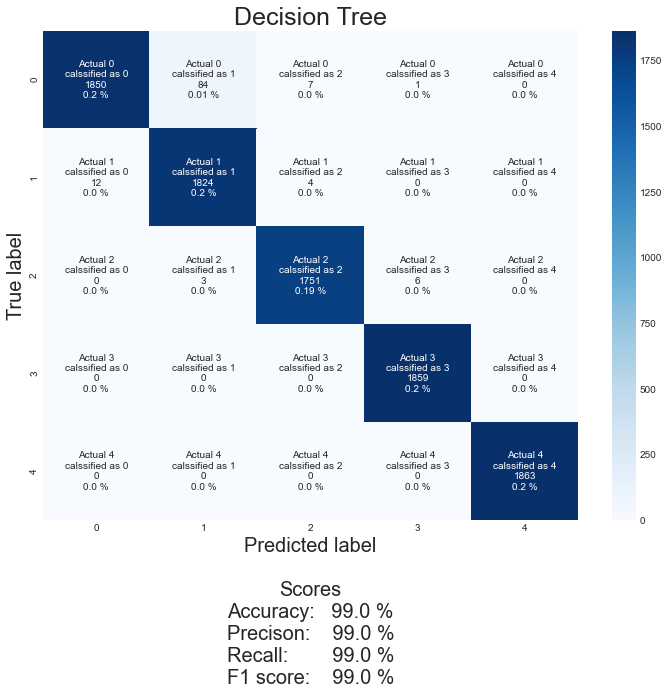


Figure Decision Tree Default

Using the standardscalar on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9923429315411422 % area under the curve.

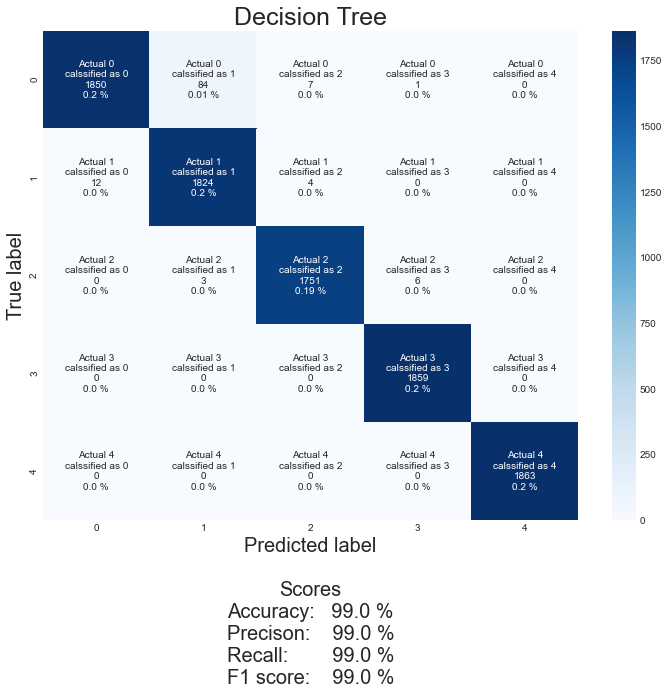


Figure Decision Tree StandardScalar

Using the MinMaxScalar on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9918932369968839 % area under the curve.

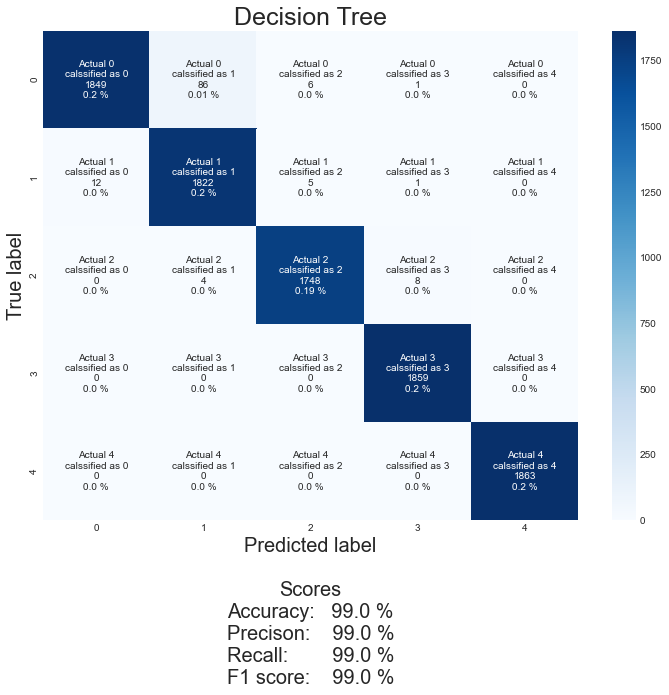
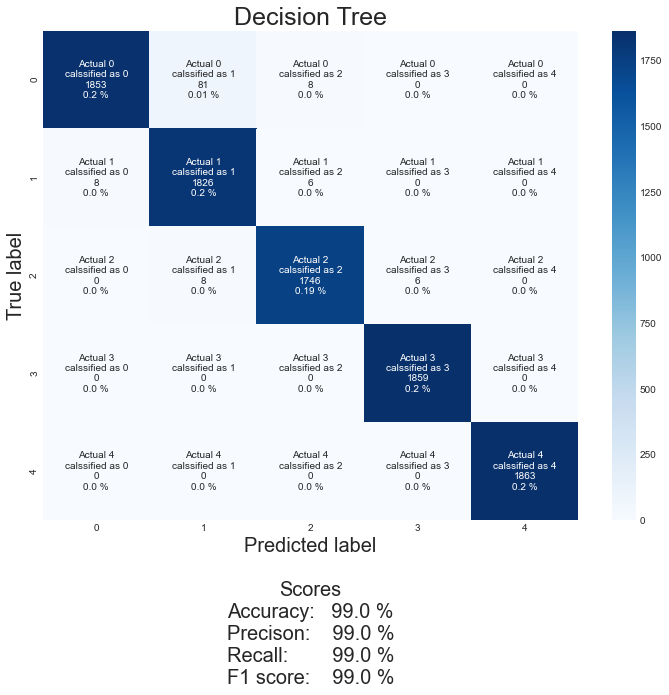
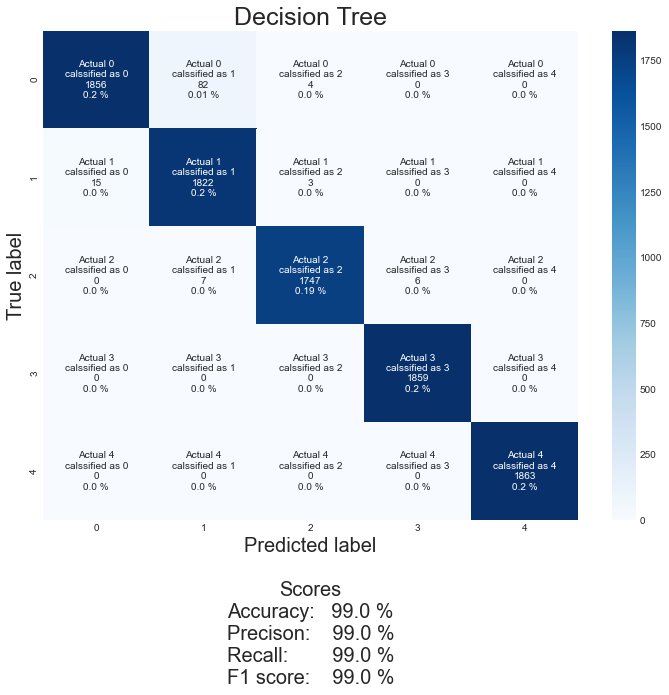


Figure Decision Tree MinMaxScalar

Using the RobustScalar on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9922851037488755 % area under the curve.



Using the Normalizer on the features, it had 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with 0.9922766918315912 % area under the curve.



with 1 min\_samples\_leaf ,2 min\_samples\_split , None max\_leaf\_nodes , 0.0 min\_weight\_fraction\_leaf , 'entropy' criterion, None max\_features, 121 max\_depth, 'random' splitter as hyperparameters all the scaling techniques give 99 % accuracy, 99 % precision, 99% recall, 99 % f1 score with MinMaxScalar has the best area under the curve is achieved so it is selected as the best scaling technique.

After cross validation of these results with the default results.

The defaults model results were 93.11 % with 0.86 % standard deviation.

The results after using minmax scaler with hyperparameter optimization were 94.44 % with 1.05 standard deviation.